VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Částečná automatizace tvorby webového uživatelského rozhraní pro aplikace založené na službách

Semi-Automated Web User Interface for Service-Based Applications

2013

Bc. Jan Deutschl

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Deutschl**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Částečná automatizace tvorby webového uživatelského rozhraní pro aplikace založené na službách
Semi-Automated Web User Interface for Service-Based Applications

Zásady pro vypracování:

Vytvořte knihovnu pro prostředí ASP.NET, která bude umožňovat semi-automatizované vytváření uživatelského rozhraní pro webové služby založené na SOAP/WSDL. Cílem je oddělit funkční a vzhledovou/uživatelskou část tvorby webové aplikace tak, aby bylo možné vytvořit funkční prototyp a ten následně vylepšovat.

1. Prozkoumejte a popište možné způsoby konzumace webových služeb pomocí .NET frameworku.
2. Navrhněte vhodné prvky webového uživatelského rozhraní pro nejčastější struktury / datové typy .NET frameworku, které se mohou ve webových službách používat jako vstup či výstup funkcí. Především pak pro hodnotové datové typy, řetězce a kolekce.
3. Vytvořte knihovnu, která bude na základě definice služby (či více služeb) poskytovat automaticky uživatelské rozhraní včetně validace vstupu.
3.1. Knihovna musí mít architekturu takovou, aby umožňovala vlastní definici nový prvků uživatelského rozhraní.
3.2. Knihovna musí umožnit (ne však vyžadovat) nastavení pro konkrétní jednotlivé funkce služby, tj. musí být možnost pro jeden z parametrů funkce použít nově vytvořený prvek uživatelského rozhraní nebo ručně zvolenou validaci.
3.3. Návrh knihovny musí umožnit ruční naimplementování jednotlivé funkce služby v ASP.NET MVC tak, aby tím nebyla poškozena funkčnost automaticky řešených funkcí.
3.4. Knihovna musí umožnit snadné vytvoření šablony funkce určující rozmístění prvků a doplnění grafické podoby. Tento proces musí být zvládnutelný osobou se znalostí HTML, CSS, JS, ale bez nutnosti programátorské úpravy.
4. Vytvořte příklad(y) demonstrující vlastnosti popsané v bodě 3.
5. Vytvořte návod pro činnost v bodě 3.4.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jakub Macek**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 7. května 2013 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2013 .....

Rád bych na tomto místě poděkoval vedoucímu práce Ing. Jakubovi Mackovi za podmětné rady a čas, který mi věnoval.

Abstrakt

Tato diplomová práce se zabývá návrhem a implementací frameworku, který slouží ke generování grafického uživatelského rozhraní na základě definice webové služby. Popisuje základní požadavky, výběr vhodné technologie a samotnou implementaci frameworku.

Klíčová slova: C#,ASP.NET, ASP.NET MVC Framework, WCF, UI

Abstract

This thesis describes the design and implementation of framework that is used to generate a graphical user interface based on the definition of web services. Describes the basic requirements, selection of appropriate technology and the actual implementation of the framework.

Keywords: C#,ASP.NET, ASP.NET MVC Framework, WCF, UI

Seznam použitých zkratek a symbolů

SOA	–	Service Oriented Architecture
SOAP	–	Simple Object Access Protocol
XML	–	Extensible Markup Language
WSDL	–	Web Services Description Language
AWS	–	Auto-Web Service Framework
HTML	–	Hyper Text Markup Language
CSS	–	Cascading Style Sheets
MVC	–	Model-View-Controller
MS-PL	–	Microsoft Public License
HTTP	–	Hypertext Transfer Protocol
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol
XSD	–	XML Schema Definition
ASP	–	Active Server Pages
ASP.NET	–	Active Server Pages .NET
URL	–	Uniform Resource Locator
IIS	–	Internet Information Services
WCF	–	Windows Communication Foundation
AJAX	–	Asynchronous JavaScript and XML
JSON	–	JavaScript Object Notation

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

S rozmachem internetu a jeho dostupností se stále více informačních technologií stěhuje do prostředí webu. Aktuální téma je také SOA (Service Oriented Architecture), tedy architektura aplikací založených na službách. Tyto aplikace jde snadno rozdělit na jednotlivé komponenty, čímž získáme robustnější kód, který lze snadněji spravovat. To je docíleno tím, že komponenty (služby na internetu) vystavují pouze svá rozhraní. A samotná komunikace je standardizovaná a platformě nezávislá.

Tato práce se zabývá návrhem aplikačního rámce (frameworku), který má sloužit ke generování grafického rozhraní k webovým službám. Cílem je usnadnit vývojářům webových aplikací práci se službami, umožnit jim služby testovat a rychle se orientovat v rozhraních, které nabízejí.

Nejdříve je popsán záměr a požadavky, které byly na implementaci kladeny. Poté se práce věnuje výběru vhodného implementačního prostředí a jeho popisem. Nakonec následuje samotný popis návrhu frameworku a jeho použití.

2 Představení záměru

Čas od času jsou vývojáři postaveni před nutnost použít webovou službu nějaké třetí strany. Stává se, že mají k dispozici kompletní dokumentaci a nebo v horším případě jen adresu služby a její WSDL schéma. To je ale pro programátora špatně čitelné.

Cílem této práce je představit nástroj, který by sloužil právě k základnímu ověření možností služby a přehlednému otestování její funkcionality. Pro tento nástroj jsem v průběhu implementace zvolil název AWS Framework (Automatized Web Service Framework). Pro zjednodušení bude v dalších kapitolách tato zkratka používána.

AWS Framework jde ale mnohem dále, než pouze k vygenerování základního grafického rozhraní pro webovou službu. Umožňuje například validaci vstupů a filtraci zobrazení vstupních parametrů a metod přes jednoduché konfigurační rozhraní založené na XML. S touto konfigurací je možné napsat jednoduchou aplikaci, která využívá jednu či více služeb s minimem vlastního kódu.

Navíc je navržen tak, aby jej každý vývojář, který má zkušenosti s vývojem aplikací založených na ASP.NET MVC Frameworku byl schopen rozšiřovat. A tak umožnit vytvořit plnohodnotnou webovou aplikaci s minimem stále se opakujících úkolů, které by bylo jinak nutné naimplementovat bez použití AWS Frameworku.

Zároveň vytváří prostředí, které může sloužit k testování samotných metod služeb, a to velmi jednoduchou cestou. Důraz byl kladen na co největší oddělení rolí vývojářů a designerů. Základní web s využitím AWS Frameworku by měl zvládnout vytvořit každý, kdo má znalosti HTML a CSS. Pro pokročilejší funkce je potom nutná znalost technologií, nad kterými je framework postaven.

Při výběru technologie, která nakonec byla použita, byly všechny tyto aspekty brány v potaz. Proto, hlavně díky oddělenosti rolí již v základní implementaci, byla vybrána technologie ASP.NET a MVC Framework. Zdrojový kód tohoto frameworku je k dispozici na internetu ([4]) pod licencí MS-PL. To bylo také považováno za velkou výhodu. Znalost kódu a vnitřní implementace pomohla k tomu, aby AWS Framework co nejvíce zapadal do již existujícího konceptu a rozšiřoval ho tak, aby všechna tato rozšíření byla v souladu s konvencemi, které jsou v MVC Frameworku použity a usnadnila tak vývojářům, kteří budou produkt používat jeho další rozšíření.

2.1 Základní vlastnosti AWS frameworku

Při návrhu a implementaci AWS Frameworku byly brány v potaz tyto aspekty:

- **Výkon** – jakákoliv obecná implementace nějakého problému, a tedy i generování grafického rozhraní pouze z definice služby, musí mít nějaké výkonnostní dopady oproti implementaci „na míru“. Je třeba je ale minimalizovat tak, aby výsledná aplikace pracovala co nejefektivněji.

- **Rozšiřitelnost** – každá funkcionalita musí být dobře zdokumentovaná a pro vývojáře jednoduše rozšiřitelná. Celková architektura musí dodržovat zavedené koncepty technologií, které využívá.

- **Snadnost použití** – výsledný rámec musí být snadno použitelný tak, aby i soba bez větší znalosti programovacích jazyků byla schopna vygenerovat základní grafické rozhraní a dělat na něm jednoduché úpravy.
- **Přizpůsobení se změnám definice služby** – změna definice webové služby nenaruší stávající funkcionalitu. Pokud ke změně dojde, automaticky generované ovládací prvky se jí okamžitě přizpůsobí.

2.2 Srovnání s jinými aplikacemi

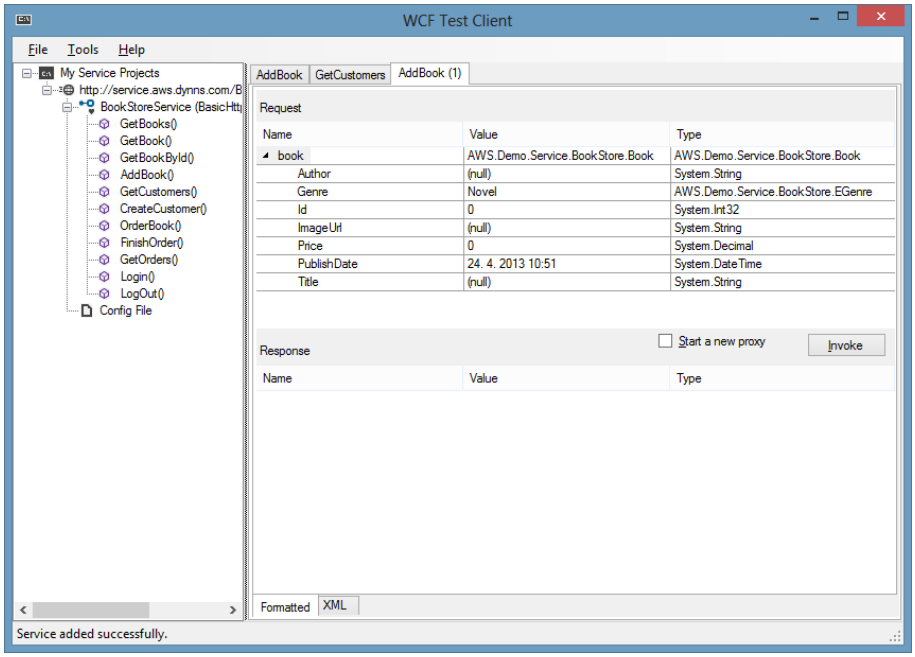
Nebyla nalezena žádná aplikace pro prostředí .NET, která na základě definice služby generuje webové rozhraní. Existují ale řešení, která slouží pouze k testování služeb jako takových, což ale není hlavním účelem popisovaného frameworku. Přesto si dvě z nich představíme.

2.2.1 WCF Test Client

WCF Test Client je jednoduchá aplikace od společnosti Microsoft. Její výhodou je přímá integrace do Visual Studia, kdy se aplikace automaticky spustí a nakonfiguruje, pokud v projektu jako startovací stránku zvolíme WCF službu.

V levé části je vidět stromová struktura obsahující seznam všech naimportovaných služeb a jejich metod. Napravo je pak detail služby, kde jsou vypsány vstupní parametru zvolené metody a je možné naplnit hodnotami. Tlačítko ”Invoke”pak slouží k vyvolání služby. Výsledek je prezentován v dolní části okna.

- Výhody
  - Integrace do Visual studia
  - Jednoduché rozhraní
  - Zdarma
- Nevýhody
  - Primárně pouze pro WCF služby
  - Pouze pro ruční testování (nelze vytvořit automatizované testy)
  - Nelze uložit nastavení a později se k němu vrátit



Obrázek 1: Aplikace WCF Test Client

2.2.2 SOAP UI

Aplikace je dostupná z `http://www.soapui.org/` ,a to v základní a profesionální verzi. Základní verze je zdarma, profesionální již je placená. Rozvržení grafického rozhraní je obdobné předchozí aplikaci. Obsahuje však více ovládacích prvků. Kromě manuálního testování aplikací umožňuje i automatické testy.

- Výhody
  - Lze vytvářet a ukládat projekty
  - Funguje pro jakékoliv SOAP služby
  - Vytváření "Test Case"na jednotlivé metody služby (automatizované testování)
  - Možnost testů výkonu serveru, kdy je vygenerováno mnoho požadavků na službu
  - Testy lze nahrát na cloudový server a spouštět je vzdáleně
- Nevýhody
  - Většina funkcí pouze v PRO verzi

3 Služby v internetu a .NET Framework

Jak již bylo řečeno v úvodu, webové služby slouží k jakési abstrakci konzumenta služby (klienta) od jejího provozovatele. Samotná komunikace není závislá na konkrétní službě a je dobře známá a zdokumentovaná pro obě strany. V současnosti je nejčastěji používaná technologie SOAP/XML.

SOAP nebo-li Simple Object Access Protocol je protokol, který slouží k výměně zpráv v prostředí internetu založených na XML. Jako protokol pro aplikační vrstvu používá nejčastěji HTTP, může ale fungovat i nad jinými (TCP, UDP). Jeho úkolem je pomocí SOAP obálky přenášet informace o volané metodě služby a její návratové hodnotě. Struktura této obálky je právě ve formátu XML. Nevýhodou použití je poměrně velká velikost obálky a tím pádem i přenášené zprávy.

Předtím, než je konzument služby schopen sestavit obálku a začít tak používat službu, musí se dozvědět o tom, které metody daná služba podporuje, jak navázat komunikaci a další informace. K tomu slouží formát WSDL (Web Services Description Language). Podobně jako SOAP i on je založen na XML. Jeho primárním úkolem tedy je popsat veřejné rozhraní služby.

Chceme-li webové služby v .NET Frameworku používat, můžeme si SOAP zprávy generovat vlastním kódem a celou logiku komunikace se službou od serializace dat, odeslání požadavku a opětovnou deserializaci vytvořit dle našich potřeb. To ale vyžaduje poměrně složitou implementaci kódu, které bychom se chtěli vyhnout. Proto se v AWS Frameworku přistoupilo k jiné variantě. A to využít nástroj svcutil.exe (ServiceModel Metadata Utility Tool)[5], který z předané definice služby (WSDL schématu) vygeneruje kód. Složité datové typy převede do tříd a vytvoří proxy třídu, tedy jakýsi vstupní bod pro volání metod služby. Výhodou je, že takto můžeme vytvořit referenci na službu, která je psaná v libovolné technologii a abstrahujeme se od SOAP protokolu a WSDL definice služby. Tento postup je navíc široce používaný a dobře rozšiřitelný. Navíc je přímo integrován do nástroje Visual Studio v němž stačí pouze kliknout pravým tlačítkem na projekt a zvolit „Add Service Reference“, čímž vyvoláme spuštění nástroje svcutil.exe s grafickým rozhraním.

3.1 Utilita SvcUtil.exe

Vygenerujeme-li odkaz na službu přímo z prostředí Visual Studia, soubory se vloží do složky „Service References\<název služby>“ v kořenovém adresáři projektu. Název služby je řetězec, který jsme při generování zadali. Do této složky se vygenerují následující typy souborů:

- **configuration.svcinfo** – XML soubor, který definuje vazbu (binding) a koncový bod (endpoint) služby. Jedná se o část konfigurace, která bývá překopírována do souboru s konfigurací (web.config případně app.config).
  - **vazba (binding)** – Definuje způsob, jakým klient se službou komunikuje. Obsahuje informace o tom, jaký protokol se použije, nastavení zabezpečení a velikosti bufferů použitých pro komunikaci.

  - **Koncový bod (endpoint)** – Obsahuje adresu služby, přidruženou vazbu k této službě a název kontraktu, který je odvozen z WSDL (na tento název se dále bude odkazovat pomocí <kontrakt>).

- <**název služby**>**.wsdl** – Překopírovaný wsdl soubor , z kterého byla reference na službu vygenerována.

- <**název služby**>**.xsd** – XSD schémata všech komplexních datových typů, které v metodách služby figurují buď jako vstupní, nebo výstupní parametry. Tyto schémata jsou pak použita při serializaci objektů do XML. Souborů typicky bývá více a jsou rozlišeny číslicí na konci (<název služby>1.xsd, <název služby>2.xsd, ...)

- **Reference.svcmap** – Zde jsou ve formátu XML uloženy informace zadané při přidávání reference na službu.

- **Reference.cs** – Samotný vygenerovaný kód v jazyce C#, který obsahuje třídy vygenerované z XSD schémat, logiku pro serializaci a deserializaci těchto tříd a rozhraní, které umožňuje volat metody služby. Kromě tříd odvozených z komplexních datových typů obsahuje tento soubor ještě dvě rozhraní a jednu třídu:

  - **Rozhraní** <**kontrakt**> - Obsahuje předpis všech metod, které daná služba vystavuje pomocí WSDL schématu.
  - **Třída** <**kontrakt**>**Client** – Třída, která implementuje předchozí rozhraní. Umožňuje volání metod služby. Dále pomocí ní lze programově definovat vazbu a koncový bod, nebo nastavit zabezpečení. Tato třída dědí z System. ServiceModel.ClientBase. V textu bude tato třída dále zmiňována jako proxy třída služby.
  - **Rozhraní** <**kontrakt**>**Channel** – Slouží pro navázání spojení se službou.

4 ASP.NET

Dříve, než se budeme věnovat MVC frameworku, je nutné si představit koncepty, nad kterými je vybudován. Z pohledu ASP.NET není totiž MVC Framework nic jiného, než handler, který zpracovává určité typy požadavků.

Technologie ASP.NET byla představena v lednu 2002 a je to nástupce klasického ASP technologie. Nyní je ve verzi 4.5. Základem pro tvorbu webu jsou takzvané WebForms. Cílem společnosti Microsoft bylo přenést principy programování aplikací známé programátorům z WinForms (Desktopových aplikací) do webového prostředí. WebForms tedy nabízejí podobnou sadu ovládacích prvků a systém událostí, které jsou vyvolány vstupem od uživatele (například kliknutím na tlačítko) jako WinForms. Technologie ASP.NET je ale natolik robustní, že umožňuje vlastní zpracování požadavků na server. To je možné implementací rozhraní IHttpHandler, které se věnuje kapitola 4.1.

4.1 Rozhraní IHttpHandler a IHttpModule

HttpHandler je třída, která implementuje rozhraní Systém.Web.IHttpHandler. Jejím úkolem je zpracovat požadavek, který přijde na server a vrátit odpověď. ASP.NET mapuje požadavky na handlery podle URL nebo přesněji přípony v této adrese. V základu jsou k dispozici čtyři druhy handlerů pro následující přípony:

- ***.aspx** – Handler pro požadavky na ASP.NET webové stránky
- ***.asmx** – Handler pro asmx webové služby
- ***.ashx** – Generický handler
- **trace.axd** – Handler, který vypisuje informace z trace na aktuální stránce

Vlastní HTTP handler musí obsahovat vlastnost IsReusable, která vrací hodnotu bool a běhovému prostředí říká, jestli může stejnou instanci handleru použít pro více požadavků a metodu ProcessRequest, která se stará o samotné zpracování požadavku.

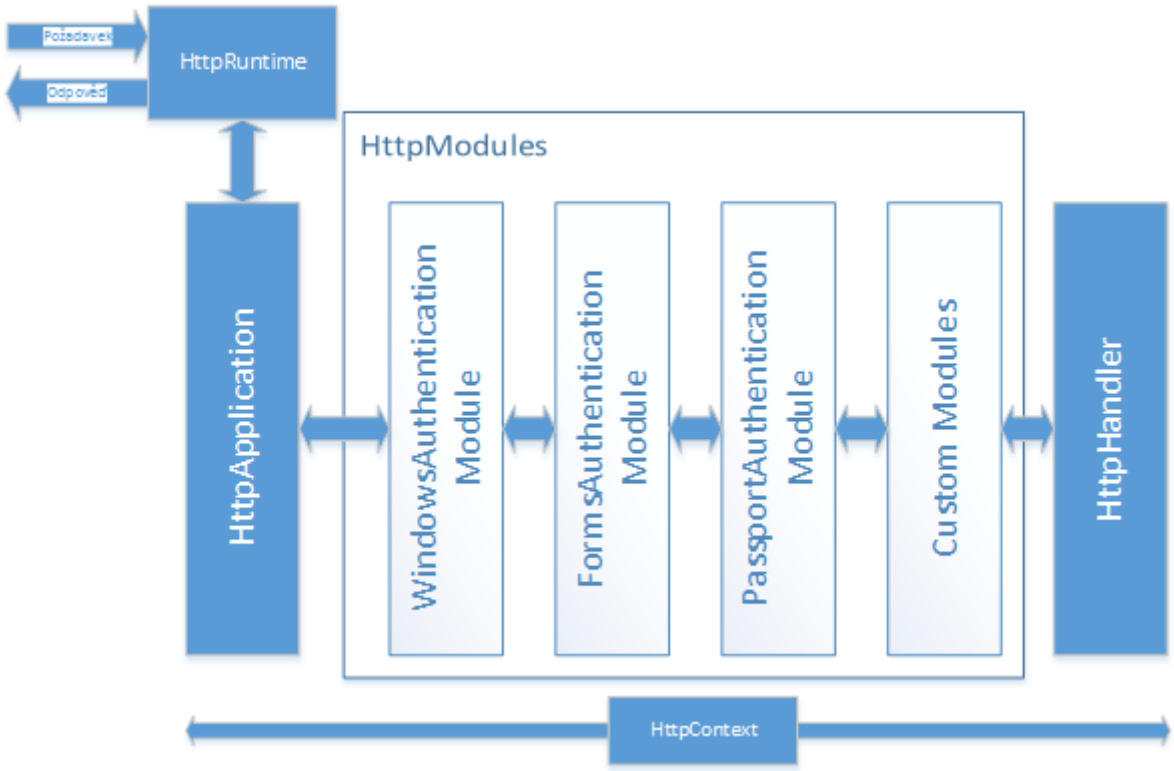
K mapování handleru na určitý typ požadavku dochází v konfiguračním souboru aplikace (soubor web.config) nebo IIS serveru (machine.config).

Obdobně lze implementovat třídu IHttpModule. Rozdíl je v tom, že oproti HttpHandleru, může jeden požadavek projít libovolným počtem modulů. Handler je vždy ale pouze jeden. Rozhraní předepisuje pouze metodu Init, v které je předán kontext HTTP aplikace. Pomocí něho se může modul zaregistrovat k reakci na různé události jako je například BeginRequest. Což událost, která nastane při začátku požadavku před vstupem do HTTP modulů.

Z obrázku 2 je vidět, že než je požadavek zpracován handlerem projde ještě všemi registrovanými HTTP moduly. Ty jsou opět registrovány v konfiguračním souboru aplikace a mohou ovlivnit průběh požadavku. Na obrázku vidíme některé z HTTP modulů, které jsou defaultně k dispozici. Je to například modul, který zajišťuje formulářovou autentizaci.

Po zaregistrování modulů se vybere na základě přípony souboru, který požadujeme (v našem případě je to .aspx) vhodný HTTP handler a zavolá se metoda ProcessRequest. Další řízení požadavku je pak již v režii příslušného handleru. Ten vrací odpověď, tak že ji zapíše do výstupního streamu v objektu HttpContext.Response.

Schéma průběhu požadavku je zobrazeno na obrázku 2. V kontextu příkladu uvedeného výše je modul HttpApplication ASP.NET a HttpHandler je handler, který je zaregistrován pro příponu souborů aspx (ve výchozím stavu to je ASP.NET Page Handler).



Obrázek 2: Průběh požadavku v ASP.NET

5 ASP.NET MVC Framework

ASP.NET MVC Framework je aplikační rámec nad technologií ASP.NET, který implementuje návrhový vzor Model-View-Controller. Byl navržen firmou Microsoft a v roce 2009 byl jeho zdrojový kód uvolněn pod MS-PL (Microsoft Public Licence). Výhodou je jednoduchost, rychlost a integrace s existujícími funkcemi ASP.NET.

5.1 Návrhový vzor MVC

Jádrem frameworku je návrhový vzor MVC, který se skládá ze tří komponent – Model, View a Controller. Popis jednotlivých komponent je v kapitole 5.2. MVC je mezi programátory dobře známý návrhový vzor a implementuje jej celá řada aplikačních rámců v různých programovacích jazycích jako například:

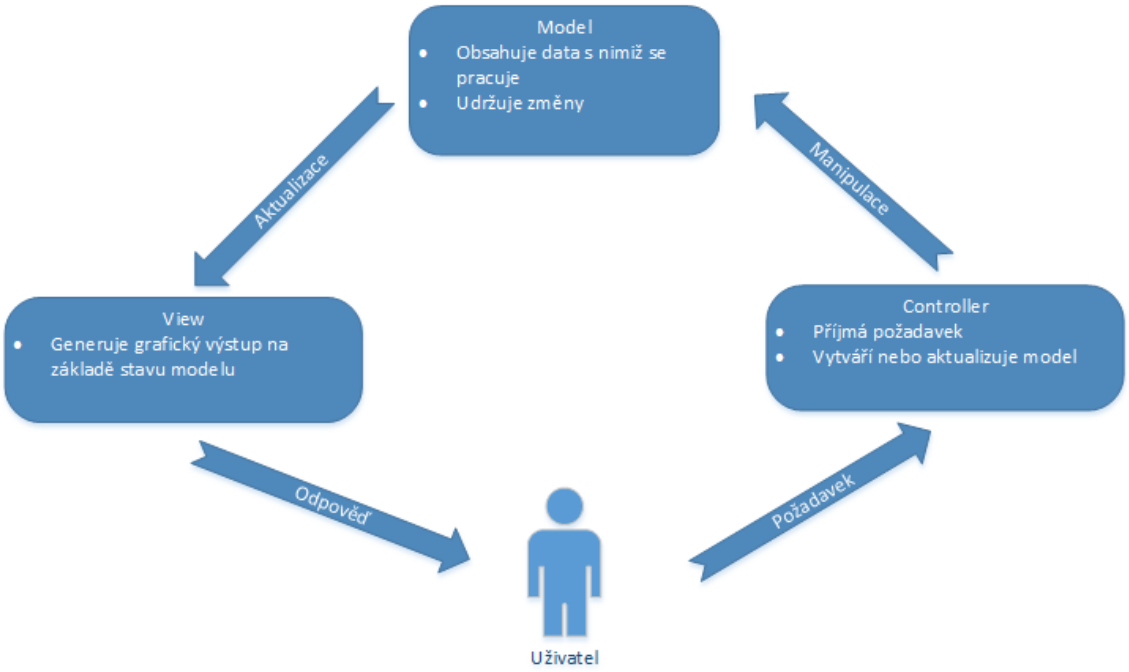
- **Java** – JavaServer Faces, Jakarta Struts a Webwork 2
- **PHP** – Nette Framework, Zend Framework
- **Ruby** – Ruby on Rails

Použití návrhového vzoru Model-View-Controller přináší některé výhody. Mezi nimi jsou:

- Jednotlivé komponenty (Model, View a Controller) jsou na sobě nezávislé a mohou fungovat samostatně.
- Díky oddělenosti jednotlivých komponent je kód přehlednější a lépe rozšiřitelný.
- Urychluje vývoj aplikací.
- Možnost testování pomocí unit testů .
- Možnost programování řízené testy (Test Driven Development).

Nyní se na MVC podíváme z pohledu webových aplikací a popíšeme si postupně akce, které se dějí od vytvoření požadavku po doručení odpovědi uživateli.

1. Uživatel inicializuje akci (zašle požadavek na server) například tak, že ve webovém prohlížeči zadá adresu.
2. Controller požadavek přijme. Podle jeho typu (uživatel chce zobrazit domovskou stránku nebo například seznam zaměstnanců firmy) získá data z datového zdroje (databáze), vytvoří model a těmito daty ho naplní.
3. Je vybrán View vhodný pro daný typ požadavku a model je mu předán.
4. View vygeneruje grafický výstup (HTML).
5. Výstup z View je poslán zpět uživateli v odpovědi na jeho požadavek.



Obrázek 3: Návrhový vzor MVC

5.2 ASP.NET MVC

Z obecné implementace MVC přejdeme do rámce ASP.NET MVC. Nalezneme jej v assembly System.Web.Mvc. Návrhový vzor MVC je zde implementován následovně:

- **Controller** – Je třída, která dědí z Systém.Web.Controller a obsahuje veřejné metody, které se v kontextu MVC nazývají akce nebo akční metody. Tyto akce jsou odlišeny svým názvem a typem požadavku, který zpracovávají. Mohou tedy existovat dvě a více akcí se stejným názvem, ale jedna například pro HTTP GET a jedna pro POST. Rozlišeny jsou atributem, který metodu dekoruje. Primárním účelem akce je inicializovat model, naplnit jej daty a vybrat View, který jej zobrazí. Každá akce musí vracet objekt typu ActionResult (proč tomu tak je, je popsáno v kapitole 5.3.3). Typicky bývají umístěny ve složce `~/Controllers`.

```
public class HomeController : Controller
{
  [HttpGet]
  public ActionResult Index(string username)
  {
    return View("Index", username);
  }
}
```

Výpis 1: Jednoduchý Controller

Podíváme-li se na výpis kódu 1, vidíme, že Controller se jmenuje Home a obsahuje jen jednu akční metodu Index. Ta jako parametr může přijmout řetězec. Akční metoda pouze vrátí objekt typu ActionResult, kde je definován název View, které se má použít a model pro něj. Metoda View() se nachází v bázové třídě Controller.

- **Model** – Slouží k předání dat mezi View a Controller. Může se jednat o jednoduchý datový typ, komplexní třídu nebo i anonymní typ. Bývá zvykem modely k jednotlivým View ukládat do složky `~/Models`.

- **View** – Soubor s příponou cshtml (vbhtml, pokud používáme Visual Basic místo C#). View je směs HTML značek a segmentů programovacího jazyka. Překlad kódu, který je ve View obsažen do výstupní čistě HTML stránky, zajišťuje tzv. ViewEngine. Těch v MVC frameworku existuje několik (například Spark, NHaml, Razor). Protože AWS využívá Razor View Engine, budeme se dále zabývat pouze jím.
  Koncept View Engine spočívá v možnosti připojitelných modulů, které definují syntaxi šablon View. [8]
  Jednoduchý příklad definice View ke zdrojovému kódu 1 v syntaxi Razor Engine by pak mohl vypadat takto:

```
<h2>Index</h2>
    @if (Model != null) {
        <b>Hello @Model !</b>
    }
```

Výpis 2: Jednoduchý View

View jsou při prvním požadavku od uživatele přeloženy do tříd a poté zkompilovány. Každé View dědí z třídy System.Web.Mvc.WebViewPage. Bývají uloženy ve složce `~/Views`. Lze je do sebe skládat a části aplikace, které se opakují (například formulář pro přihlášení uživatele), mít tak na jednom místě. Tyto částečné View bývá zvykem ukládat do složky `~/Views/Shared`.

5.3 Průběh požadavku v ASP.NET MVC

Celý proces zpracování požadavku v ASP.NET MVC Frameworku se skládá z více kroků. Tyto kroky si můžeme rozdělit do pěti fází:

- **Směrování (Routing)** – Hlavním účelem je na základě URL vytvořit směrovací data, která jsou pak v průběhu požadavku dále použita.

- **Vytvoření Controlleru (Controller Creation)** - V této fázi se ze směrovacích dat zjistí, který Controller se má použít a vytvoří se jeho instance.

- **Vyvolání akce (Action Execution)** - Podle typu požadavku a vstupních parametrů se vybere vhodná akce instance třídy Controller z předchozí fáze. Vyvolá se akční metoda, která má typicky za úkol získat data, vytvořit model a vybrat View, který se použije pro renderování modelu.
- **Renderování View (View Render)** – Poslední fáze vyrenderuje HTML stránku z definice modelu a view, který jí byl předán z předchozí fáze.

Do tohoto procesu má programátor možnost vstoupit a ovlivnit jej na mnoha úrovních. Toho bylo využito při implementaci AWS Frameworku tak, aby logika vytvářeného rámce co nejvíce zapadala do tohoto konceptu a dále jej rozšiřovala. Z tohoto důvodu si jednotlivé fáze požadavku v HttpHandleru MVC Frameworku popíšeme podrobněji. Akce, které se provedou před vstupum / po vstupu do handleru, jsou uvedeny v kapitole 4.

5.3.1 Směrování (Routing)

Směrování slouží k mapování požadavku na konkrétní Controller a jeho akci. Po vytvoření nové MVC aplikace v prostředí Visual Studia se automaticky zaregistruje jedno mapování (Registrace cest probíhá při startu aplikace v souboru Global.asax).

```
public static void RegisterRoutes(RouteCollection routes)
    {
      routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
      routes.MapRoute(
          name: "Default",
          url: "{controller}/{action}/{id}",
          defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
      );
    }
```

Výpis 3: Ukázka směrování

Výpis zdrojového kódu 3 přidává do kolekce RouteCollection novou hodnotu. To se děje metodou MapRoute, která má tři parametry. Jednoznačný název cesty, URL, kde ve složených závorkách můžeme definovat měnící se parametry a nakonec defaultní hodnoty těchto parametrů. Parametry controller a action mají speciální význam, protože určují konkrétní Controller a akci, která se zavolá. Id je pak volitelný parametr, který se předá metodě (akci) pokud jej deklaruje.

Defaultní hodnoty nám také říkají, že adresa „˜/“ je shodná s adresou „˜/Home/Index“ a parametr Id je volitelný. Například adresa `~/Home/Index/5` by opět vyvolala akční metody Index, ale předala by jí jako parametr číslo 5.

V registrování cest lze jít ještě dále a definovat tzv. constraints, neboli omezení. Ty pak na základě regulérního výrazu, nebo implementace rozhraní IRouteConstraint mohou ovlivnit, která cesta se vybere. V implementaci AWS Frameworku však nejsou použity, takže se jimi podrobněji nebude tato práce zabývat.

Samotný výběr cesty pak spočívá v iteraci RouteCollection, což je kolekce tříd, které dědí z abstraktní třídy RouteBase a do které se cesty vkládají. Na každý objekt v této

kolekci je pak zavolána metoda GetRouteData, dokud tato metoda nevrátí výsledek (RouteData). Je tedy zřejmé, že záleží na pořadí, ve kterém se cesty do kolekce registrují.

5.3.2 Vytvoření Controlleru (Controller Creation)

V této fázi se vytvoří instance konkrétní třídy Controller na základě výsledku z předchozího kroku, který je k dispozici v objektu RouteData. Controller se získá zavoláním metody CreateController instance třídy, která implementuje rozhraní IControllerFactory a je dostupná přes statickou vlastnost ControllerBuilder.Current.GetControllerFactory().

5.3.3 Vyvolání akce (Action Execution)

Nyní je k dispozici instanci třídy Controller a v RouteData máme uloženy informace z routovacího modulu. Dalším krokem tedy je vyvolat konkrétní akční metodu. Důležitou informací z routovacích dat je název akce a o jaký HTTP požadavek se jedná (GET, POST, DELETE. . .). O samotné nalezení akce se stará třída ControllerActionInvoker. Její konkrétní instance je vytvořena v inicializaci samotné třídy Controller a je dostupná přes její vlastnost ActionInvoker. Rozhraní IActionInvoker, které tato třída implementuje, jí předepisuje pouze jednu metodu a to FindAction. Ta na základě názvu akce a popisu třídy Controller (instance třídy ControllerDescriptor) nalezne vhodnou akci a vrátí instanci objektu ActionDescriptor.

Předpokládejme nyní, že byla nalezena vhodná akční metoda. Poté třeba pokusit se z dostupných dat vytvořit parametry pro její spuštění. Ty se dají získat z routovacích dat. Například akční metoda může přijímat parametr id, tak jak byl uveden ve výpisu kódu 1. Další možností je pak, pokud jde o formulář (HTTP požadavek POST), předat akční metodě parametry z hodnot ve formuláři. To lze udělat dvěma způsoby. Buď metodě předat kolekci FormCollection, nebo přímo konkrétní model, který byl použit pro vytvoření formuláře. Výpis zdrojového kódu 4 ukazuje, že druhá varianta je jistě přehlednější. Navíc odpadnou starosti s přetypováním a validací dat, které se věnuje kapitola 5.4.

```
[HttpPost]
 public ActionResult Index(FormCollection form)
 {
   Book book = new Book()
   {
     Author = form["Author"],
     Price = Convert.ToDecimal(form["Price"]),
     PublishDate = Convert.ToDateTime(form["PublishDate"]),
      Title  = form[" Title "]
   };
   book.SaveToDb();
   return View("BookSaved");
 }

 [HttpPost]
 public ActionResult Index(Book book)
 {
   book.SaveToDb();
```

```
        return View("BookSaved");
    }
```

Výpis 4: Předávání parametrů akčních metod Controlleru

Má-li být akční metodě předána instance modelu, musí být nejdříve někde vytvořena. K tomu se v MVC Frameworku používá konstrukce nazvaná ModelBinder. Ta má za úkol z dostupných dat v požadavku vytvořit instanci objektu požadovaného akční metodou a naplnit ho daty. Tyto data získává z kolekce ValueProviders. Tato kolekce obsahuje nejčastější zdroje dat z požadavku, Jsou jimi:

- Data z formuláře (Request.Form).
- Parametry získané v předchozí akční metodě (v případě že akce je potomek jiné akce - metoda je dekorována atributem ChildActionOnly).
- Data ze směrování (RouteData.Values)
- Pokud se jedná o AJAX požadavek, tak JSON data z těla tohoto požadavku.
- Parametry z URL (Request.QueryString).
- Soubory, které může obsahovat požadavek POST (Request.Files).

Kolekci zdrojů dat lze rozšiřovat implementací třídy ValueProviderFactory a přidáním této implementace do ValueProviderFactories.Factories.

Samotný ModelBinder pak je třída, která implementuje rozhraní IModelBinder. Toto rozhraní předepisuje jedinou metodu BindModel. K volání této metody dojde před vstupem do akční metody a jejím účelem je vytvořit instanci modelu na základě dat dostupných z požadavku. Typicky v ní také proběhne validace.

Ještě před samotným vyvoláním akce přijdou na řadu ActionFilters. Tyto třídy podobně jako ActionInvoker slouží programátorům ke vstupu do pipeline a jejího ovlivnění. Třída ActionFilter musí dědit z abstraktní třídy ActionFilterAttribute, která obsahuje čtyři virtuální metody. I když jsou metody virtuální a ne abstraktní, jejich implementace v base třídě je prázdná z důvodu toho, aby programátor mohl přepsat pouze metody, které jej zajímají.

- **OnActionExecuting** – zavolá se těsně před vyvoláním akční metody controlleru
- **OnActionExecuted** – zavolá se po dokončení akční metody Controlleru
- **OnResultExecuting** – zavolá se před renderováním View
- **OnResultExecuted** – zavolá se po vyrenderování View

ActionFilter se používá jako atribut a je deklarovaný v definici třídy nebo konkrétní akce ve třídě Controller. Typickým využitím může být například autorizace uživatele k použití konkrétní akční metody.

Posledním krokem v této fázi je samotné vyvolání akční metody. Její deklarace je zcela v rukou programátora a účel této metody již byl popsán dříve. Po dokončení se ještě zavolají zaregistrované ActionFilters a výstup z akční metody (ActionResult) se pošle k vyrenderování do View.

5.3.4 Renderování View (View Render)

V této fázi již je k dispozici Model a název View, který se má použít k jeho renderování. Tyto informace jsou k dispozici v objektu ActionResult, který je předán z předchozí fáze.

Nejdříve dojde k volání ActionFilters (metody OnResultExecuting), poté přijde na řadu ViewEngine. Ten má za úkol najít příslušný View a vytvořit jeho instanci. Aplikace mohou obsahovat více ViewEngine a zároveň ViewEngine hledá na různých místech. Použije se první shoda. Řekněme, že požadujeme View „MyIndex“, na který se odkazujeme z třídy Controller s názvem „HomeController“. Defaultní ViewEngine pak prohledá tyto lokace:

- ~/Views/Home/MyIndex.aspx
- ~/Views/Home/MyIndex.ascx
- ~/Views/Shared/MyIndex.aspx
- ~/Views/Shared/MyIndex.ascx
- ~/Views/Home/MyIndex.cshtml
- ~/Views/Home/MyIndex.vbhtml
- ~/Views/Shared/MyIndex.cshtml
- ~/Views/Shared/MyIndex.vbhtml

Jde vidět, že do hledání jsou zapojeny i klasické ASP.NET stránky (.aspx, .ascx) a není problém v jedné aplikaci používat jak je, tak i MVC Framework. Pokud ani v jedné z prohledávaných lokací engine nenajde vhodný View, vyvolá výjimku System.InvalidOperationException, v opačném případě vytvoří instanci třídy s odpovídajícím View.

Na stránku se přidá javascriptová logika pro validaci na straně klienta a zavolají se helper metody (viz. Kapitola 5.5), které vygenerují zbylý HTML kód. Nakonec dojde ještě k volání ActionFilters metod OnResultExecuted. Tím celý průchod handlerem končí a výsledek je odeslán zpět do klientského prohlížeče.

5.4 Validace vstupu v ASP.NET MVC

Validace vstupů je důležitou součástí každé aplikace. Pomáhá zajišťovat konzistenci dat ještě před jejich vložením do databáze nebo jiného perzistentního úložiště. Samotná validace probíhá spouštěním většinou podobných úloh nad daty, které mají prověřit, zda

uživatel zadal korektní informace. Z tohoto důvodu bývá často integrována přímo do aplikačního rámce. Stejně tomu je i v MVC frameworku.

Zde opět dobře poslouží oddělenost datových závislostí, které je dosaženo díky použití MVC návrhového vzoru. To, co chceme validovat je pouze Model, přesněji jeho veřejné vlastnosti (properties). Nejjednodušší cestou, která zároveň dobře dodržuje konvence MVC, jsou DataAnnotations. Jedná se o třídy, které jsou implementovány jako atributy a dekorují se jimi vlastnosti modelu, které chceme validovat. Nalezneme je v assembly System.ComponentModel.DataAnnotations. Jako příklad poslouží jednoduchý model entity kniha na příkladu kódu 5

```
public class Book
 {
   [Required]
   [StringLength(50,MinimumLength=10)]
   public string Title { get; set; }

   [Required]
   [StringLength(50, MinimumLength = 10)]
   public string Author { get; set; }

   [Required]
   [Range(1,1000)]
   public decimal Price { get; set; }

   public DateTime? PublishDate { get; set; }

   public void SaveToDb()
   {
     // save this entity to DB
   }
 }
```

Výpis 5: Model entity kniha

Jak vidíme ze zdrojového kódu 5, entita kniha má vlastnosti Název, Autor, Cena a Datum vydání. Ty jsou dekorovány atributy, které na ně kladou integritní omezení Například autor je povinná položka a musí mít nejméně 10 znaků. DataAnnotations využívají i jiné aplikační rámce a technologie. Nás ale zajímá jak s nimi pracuje MVC Framework.

V průběhu vytváření instance třídy Controller vznikne i kolekce ViewData. Ta obsahuje mimo jiné i jednu pro validaci velmi důležitou vlastnost, a to je ModelState. Jak název napovídá ModelState (přesněji ModelStateDictionary) je třída, která má za úkol udržovat stav modelu. Jedná se o kolekci Dictionary, která má jako klíč řetězec a hodnotu objekt typu ModelState. V kapitole 5.3.3 byl diskutován ModelBinder a v něm právě dochází k vytváření instance modelu a tím pádem rovnou i k validaci vlastností tohoto modelu. Pokud dojde k chybě (validace neprojde), je informace o tomto stavu uložena právě ve výše zmíněné kolekci. Programátor má pak dále možnost v těle akční metody provést validaci ručně tím, že jednoduše do kolekce ModelState přidá další položky. Ta je pak

předána do fáze, kdy se vytváří View. Ta má možnost při generování grafického výstupu na chyby v ModelState reagovat a upozornit uživatele chybovou hláškou.

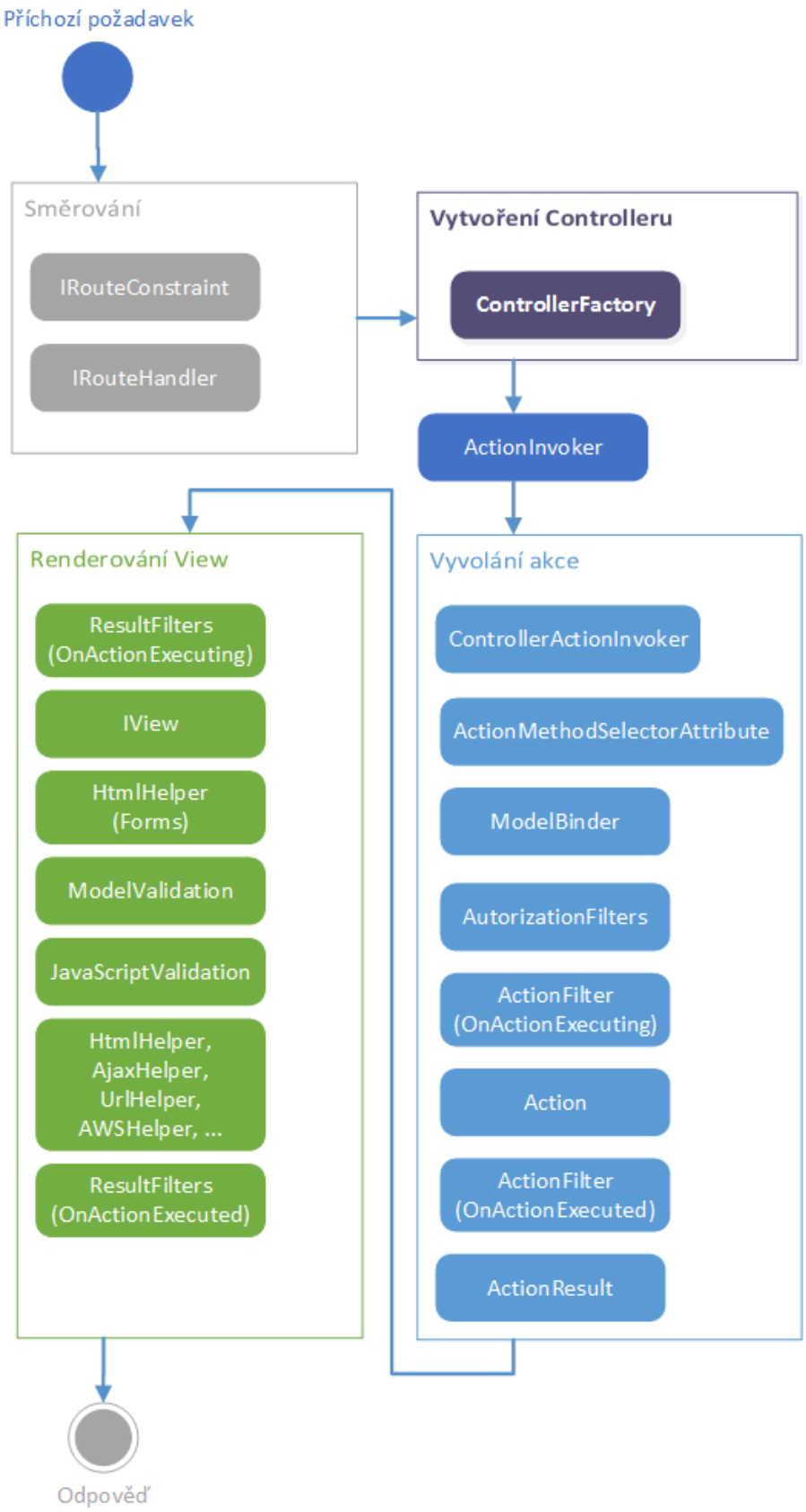
5.5 Pomocné metody (Helper methods)

Sada těchto metod je nejčastěji využívána ve View pro generování HTML kódu, který je pak odeslán na klientský prohlížeč. Úkolem těchto metod je usnadnit a zobecnit základní kroky, s kterými by se jinak programátor musel neustále dokola potýkat.

Nejpoužívanější třídu - HtmlHelper najdeme přímo v namespace System.Web.Mvc. Tato třída však neobsahuje kompletní sadu nejpoužívanějších metod, ale jen jakýsi základ, nad kterým budují další metody svou funkcionalitu. Většina jich je totiž implementována jako Extensions Methods.

Extensions Methods, neboli rozšiřující metody, umožňují již k existujícímu typu (třídě) přidat další funkcionalitu bez nutnosti dědit z této třídy nebo ji modifikovat. Jsou k dispozici od C# 3.0. V podstatě se jedná o statickou třídu (třída musí být deklarována jako statická), která obsahuje opět statické metody. To, který typ daná metoda rozšiřuje, určuje vždy její první parametr, který předává právě instanci tohoto typu do těla metody a před nímž musí být klíčové slovo this.

Tyto rozšiřující metody představují elegantní způsob jak základní třídu HtmlHelper rozšířit a také se takto používají. Celá jejich řada je v namespace System.Web.Mvc.Html. Najdeme zde nejrůznější metody pro základní prvky jako Label, RadioButton, CheckBox a další. Jejich výstupem je instance typu MvcHtmlString a o jejich spuštění v rámci View se stará již zmíněný ViewEngine.



Obrázek 4: ASP.NET MVC Framework pipeline

6 AWS Framework

6.1 Požadavky AWS

Při použití AWS Frameworku pro tvorbu webové aplikace je nutné splňovat minimální požadavky, které jsou:

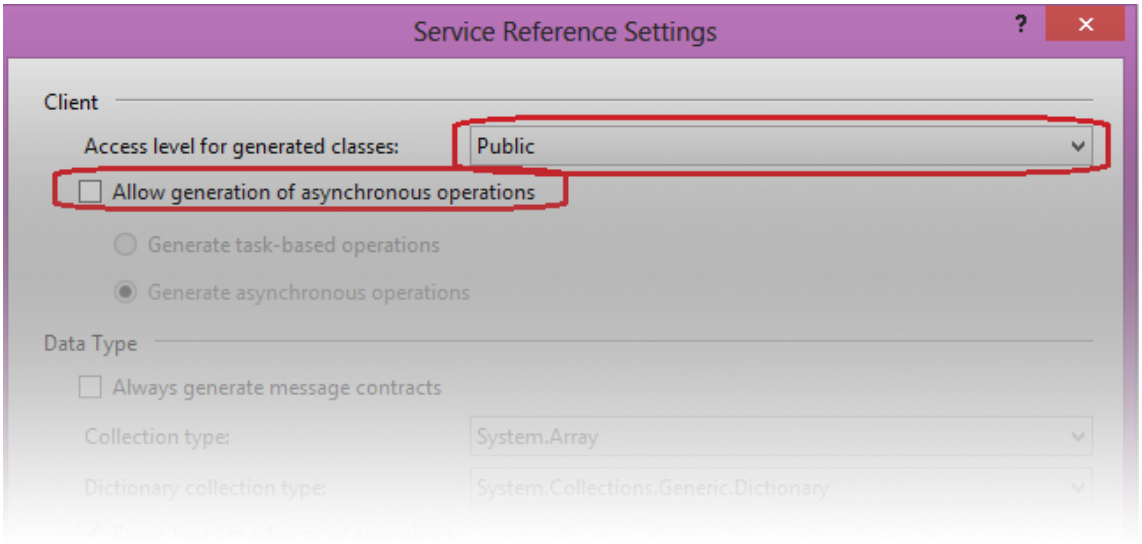
- Microsoft Visual studio 2010 a vyšší
- Nainstalovaný .NET Framework 4.0 a vyšší
- Nainstalovaný ASP.NET MVC Framework verze 3 a vyšší
- Knihovna AWS.Web.Core
- Knihovna AWS.Service.Proxy
- JQuery 1.7.1 a vyšší (není podmínkou)
- JQuery UI 1.8.20 a vyšší (není podmínkou)

Jsou-li tyto požadavky splněny, je možné začít Framework používat. Jeho konfigurací a nastavením se budou věnovat další kapitoly.

6.2 Jak začít AWS Framework používat

Předpokládejme nyní, že jsou podmínky z kapitoly 6.1 splněny, máme v nástroji Visual Studio vytvořený nový projekt typu ASP.NET MVC Web Application a známe adresu webové služby, kterou chceme pro naši aplikaci použít. Dále je nutné následovat tyto kroky:

1. Do projektu typu ASP.NET MVC Web Application přidáme referenci na knihovnu AWS.Web.Core.
2. Způsobem, který byl diskutován v kapitole 3.1, vytvoříme referenci na službu. Při vytváření služby je nutné ve formuláři Service reference Settings nechat přednastavené hodnoty (viz. obrázek 5):
    - Access level for generated classes na public. To z toho důvodu, že vygenerované třídy a jejich metody musí být přístupné knihovně AWS.Web.Core
    - Allow generation of asynchronous operations na false. Z povahy aplikace nemá smysl generovat asynchronní metody pro komunikaci se službou.



Obrázek 5: Nastavení reference na službu

3. Posledním krokem je přidání Controlleru do aplikace, který bude metody služby zobrazovat. Tento Controller musí dědit z třídy AWSController, jejíž generické atributy předávají informace o službě, která se má použít. Podrobným popisem této třídy se zabývá kapitola 8.1

4. Po prvním spuštění aplikace se vytvoří konfigurační soubor aws.config, který se uloží do hlavní složky projektu s webem (`~/aws.config`). V tomto konfiguračním souboru je možné upravovat některé vlastnosti webové stránky. Viz kapitola 7.3.

6.3 Průběh požadavku v AWS Frameworku

V této kapitole bude diskutován průběh požadavku, který obslouží AWS Framework z pohledu MVC. Kapitola se věnuje pouze nastínění základních kroků, kterými požadavek prochází tak, jak je znázorněno na obrázku 6. Každému tomuto kroku se pak věnuje samostatná kapitola dále v textu. Jak dojde k samotnému volání služby, validaci vstupů a vrácení výsledku bude také vysvětleno v následujících kapitolách.

První částí, do které AWS Framework zasahuje, je výběr akční metody třídy Controller. Každá metoda služby může mít vlastní akční metodu. Pokud ji ale nemá naimplementovanou, zavolá se defaultní akce, která je součástí třídy AWSController. Na obrázku 6 vidíme, že výběr akční metody probíhá v bloku AWSControllerActionInvoker. Třídy odvozené z ControllerActionInvoker byly popsány v kapitole 5.3.3. Tato konkrétní implementace hledá, jestli ve třídě Controller existuje akční metoda pro danou metodu služby a pokud ne, tak vyvolá defaultní akci.

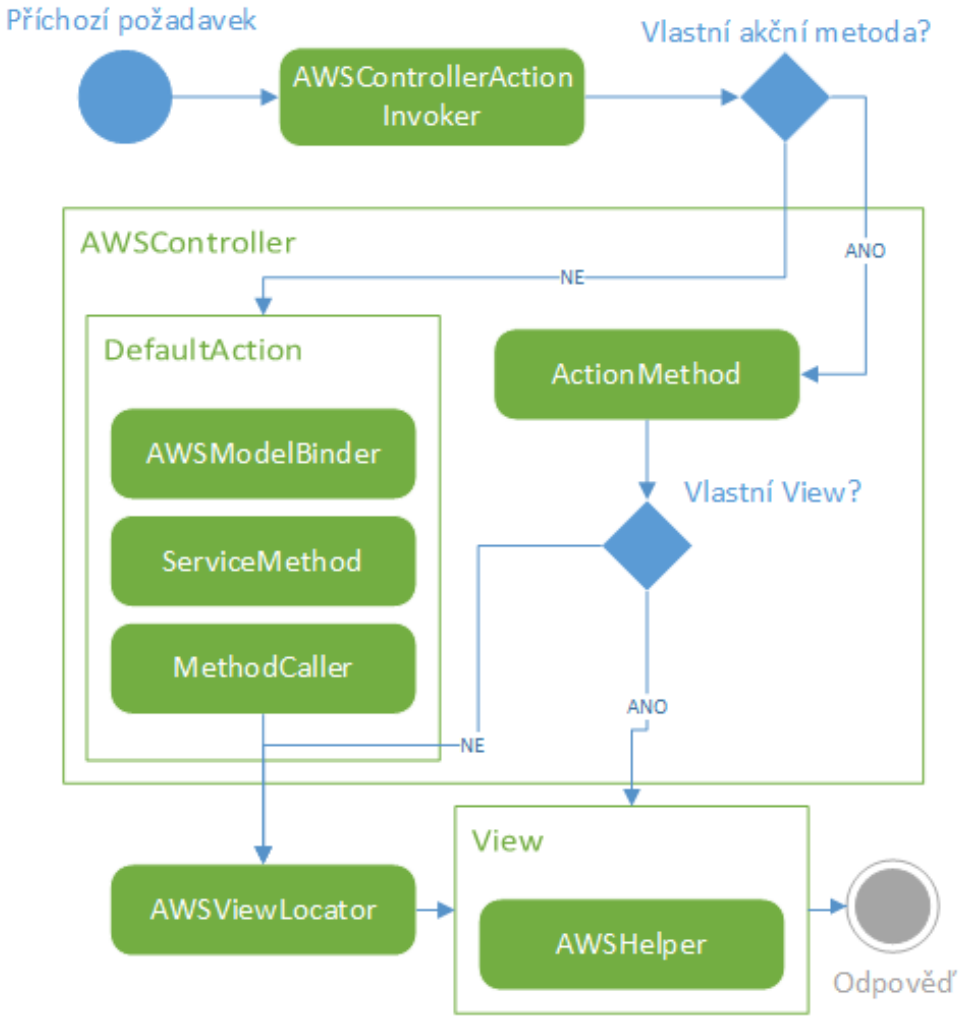
Vlastní akční metodu píše vývojář (uživatel AWS Frameworku). Umožňuje mu převzít absolutní kontrolu nad průběhem požadavku. Ve vlastní akční metodě je možné

přizpůsobit logiku validace, vytváření reference na metodu služby a její zavolání a zpracování výsledku. Nakonec se může vývojář rozhodnout, jestli pro renderování stránky použije vlastní View, nebo nechá na AWS Frameworku, aby vygeneroval defaultní View podle šablony.

Pokud není implementována vlastní akční metoda, projde požadavek defaultní akcí, která vytvoří referenci na metodu služby a pokud má k dispozici dostatek parametrů proběhne validace a samotné zavolání služby.

V následující části již jsou k dispozici všechna data a stačí zavolat příslušné View a vyrenderovat výsledek. Defaultní akční metoda vždy používá třídu AWSViewLocator (nebo třídy z ní odvozené). Jak její název napovídá úkolem této třídy je na základě známých informací o metodě služby, vybrat vhodné View. Opět je k dispozici základní implementace a možnost pro vývojáře si ViewLocator naimplementovat podle vlastních představ (kapitola 8.6). Zjednodušeně ViewLocator funguje tak, že pokud například výsledkem volání služby je nějaká kolekce dat, předpokládá, že je má zobrazit jako tabulku. A na základě tohoto předpokladu vybere jeden z View (šablon) o kterých ví, že má pro renderování výsledku k dispozici.

V poslední řadě AWS Framework vstupuje do samotného generování HTML kódu ve View. A to tak, že rozšiřuje standardní HTML Helper o takzvaný AWSHelper. Jeho účelem je zjednodušit generování výsledku na základě definice metody služby. Podrobný popis tohoto helperu i se všemi dostupnými parametry je v kapitole 8.7.



Obrázek 6: Průběh požadavku v AWS Frameworku

6.4 Rozdělení projektu do dvou částí

Celý AWS Framework je rozdělen do dvou samostatných knihoven. A to:

- **AWS.Web.Core.dll** – Hlavní funkce této knihovny je generovat grafické rozhraní a spolupracovat s ASP.NET MVC Frameworkem. Je tedy vhodná pouze pro použití s webovými aplikacemi.
- **AWS.Service.Proxy.dll** – Naopak tato knihovna slouží pouze ke komunikaci se službou. Umožňuje konfiguraci, vytváří a udržuje spojení se službou a vystavuje rozhraní, které se dá použít pro volání metod služeb a získávání informací o těchto metodách.

Výhodou tohoto rozdělení je, že druhá knihovna není závislá na použití pro webové aplikace a dá se použít například pro automatické generování unit testů pro jakoukoliv službu na internetu, nebo v jakékoliv jiné než webové aplikaci.

7 Část služby (AWS.Service.Proxy)

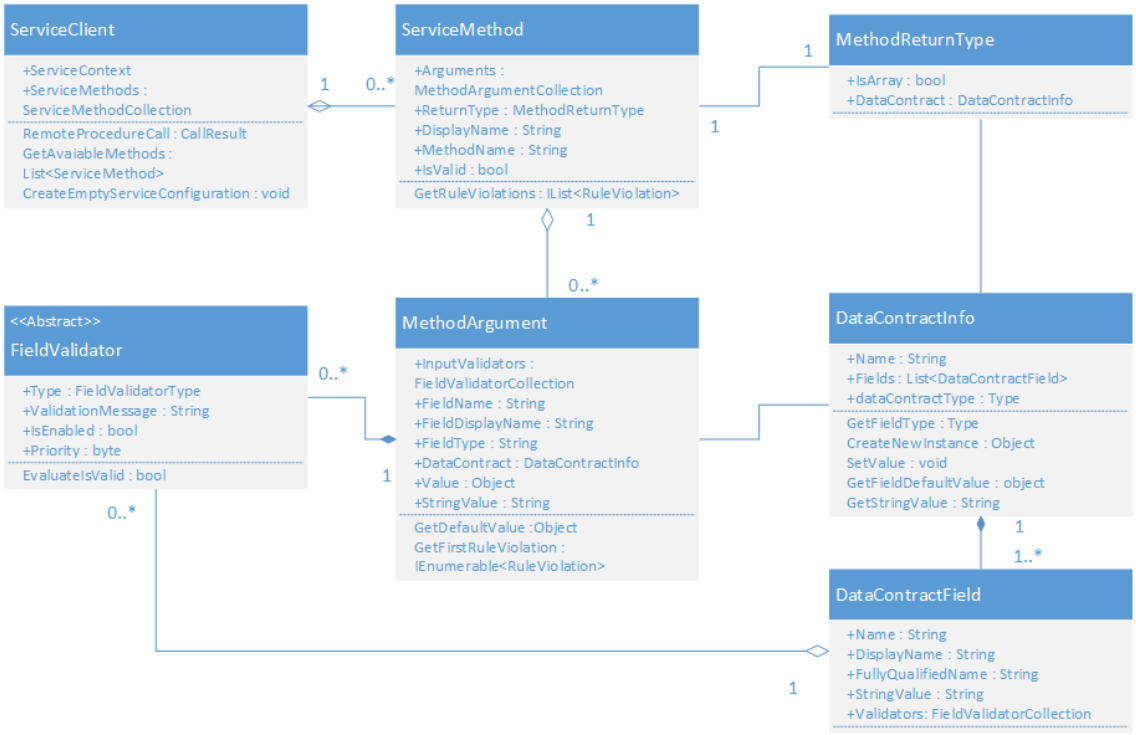
Tato knihovna má za úkol komunikovat se službou a poskytovat jiným částem aplikace informace o metodách této služby. Základním prvkem je instance třídy ServiceClient. Ta je vždy vytvořena v kontextu konkrétní služby a je jí předána instance proxy třídy této služby (výrazu proxy třída se věnuje kapitola 3.1).

ServiceClient umožňuje získávat informace o metodách služby, které udržuje v kolekci ServiceMethods a samotné volání služby a navrácení výsledku.

Metoda služby je reprezentována třídou ServiceMethod. Každá metoda obsahuje:

- **Název** – Jednoznačná identifikace metody v rámci služby (WCF služby nedovolují přetěžování názvů metod v rámci jednoho kontraktu).
- **Vstupními parametry** – tedy kolekci argumentů, které se zadávají při volání metody služby. Tyto jednotlivé argumenty mohou být jak jednoduchým datovým typem, tak i komplexním typem (třídou). Komplexní datové typy jsou reprezentovány třídou DataContractInfo. Každý argument metody může mít na sobe navázány validátory. O validaci vstupů pojednává kapitola 7.4.
- **Výstupní typ** – Opět může být jak jednoduchý, tak komplexní datový typ. Je reprezentován třídou MethodReturnType.

Konkrétní implementaci tohoto uspořádání, tak jak je v knihovně AWS.Service.Proxy, znázorňuje třídní diagram na obrázku 7. Pro přehlednost jsou do diagramu zahrnuty pouze veřejné vlastnosti a metody. Množina všech validátorů je reprezentována abstraktní třídou, kterou konkrétní typy validátorů implementují.



Obrázek 7: Třídní diagram AWS.Service.Proxy

7.1 Práce se vstupními a výstupními parametry metod služby

Jsou rozlišovány dva typy vstupních a výstupních parametrů.

- **Jednoduché datové typy** - string, enum, double, DateTime, Int16, Int32, Int64, UInt16, UInt32, UInt64, Decimal, Fload, Bool a generický typ Nullable<T>, kde T je jeden z předcházejících typů.
- **Komplexní datové typy** - V tomto případě se jedná o třídu, která musí být složena z jednoduchých datových typů. Definice takovéto třídy je dále v této práci nazývána datovým kontraktem (DataContract).

Hodnoty parametrů volané metody jsou předány do kolekce Arguments ve třídě ServiceMethod pomocí volání metody SetValue nebo SetDataContract.

Metoda SetValue přijímá odkaz na argument metody, jehož hodnotu má nastavit a nastavovanou hodnotu. Ta je typu Object a v těle této metody je přetypována na konkrétní typ vstupního parametru. Pokud se přetypování nezdaří je vyvolána výjimka InvalidArgumentCastException. Ta obsahuje informace o argumentu a chybu, která při konverzi nastala.

Metoda SetDataContract slouží k nastavení vstupního parametru, který je komplexním typem (třídou). Existují tři přetížené varianty této metody. všechny jako první para-

metr přijímají odkaz nastavovaný argument metody. První dvě získávají data z předané kolekce typu NameValueCollection. Třetí přetížené metodě není předána kolekce, ale delegát funkce (struktura Func<string,object>). Ten je deklarován tak, že přijímá jeden parametr typu string a vrací obecný objekt. Delegát je pak zavolán vždy, když se získává hodnota pole pro datový kontrakt.

Pokud se nepodaří získat hodnotu pro některé z polí datového kontraktu nastaví se tato hodnota na výchozí. V případě, že selže konverze typu je vyvolána výjimka InvalidDataContractArgumentCastException.

Po nastavení hodnot argumentů služby jsou tyto hodnoty udržovány ve vlastnosti Value každého argumentu metody.

7.2 Vzdálené volání metod služby

Volání metod služby se provádí přes třídu ServiceClient. Té je v konstruktoru předána instance proxy třídy služby, kterou chceme používat. Samotné zavolání se provede přes metodu RemoteProcedureCall. Ta přijímá jako svůj parametr konkrétní definici metody služby, která se má volat.

Návratovým typem metody RemoteProcedureCall je třída CallResult. Obsahuje čtyři vlastnost, kterými jsou:

- **Result** - typu object - Tedy výsledek volání vzdálené metody nebo null, pokud k němu nedojde, nebo metoda nevrací žádný výsledek.
- **Error** - typu Exception - Obecná výjimka, která může nastat při volání vzdálené metody na straně serveru. Pokud k výjimce dojde, je zachycena a vložena do této vlastnosti, aby s ní vývojář mohl dále pracovat v UI vrstvě.
- **IsValid** - typu Boolean - indikuje, zda vstupní parametry vzdálené metody prošli validačním procesem. Pokud alespoň jeden validátor neprošel, je tento příznak nastaven a volání metody je ukončeno.
- **RuleViolations** - typu IList<RuleViolation>- kolekce chybových hlášení. Každá položka této kolekce má dvě vlastnosti. Název vstupního parametru, pro který validace neprošla a zprávu, která se má zobrazit v UI.

7.3 Konfigurace

Konfigurační soubor je vytvořen při prvním startu aplikace (pokud již neexistuje) na základě XSD schématu, které je přiloženo v příloze A této práce. Vygeneruje se ze všech známých referencí na služby, které byly do projektu přidány a vloží se do kořenového adresáře projektu. Název souboru je aws.config.

Pomocí tohoto souboru může vývojář konfigurovat:

- Pro každou službu (**Service**)
  - **DisplayName** - zobrazovaný název služby

- Pro každou metodu (**MethodInfo**)
  - **Enabled** - jestli je povolena a tedy zobrazí-li se ve výpisu metod
  - **DisplayName** - její zobrazovaný název
- Pro každý návratový typ metody (**MethodReturnType**)
  - **DataContract/DataContractName** - název datového kontraktu nebo datový kontrakt
- Pro každý vstupní parametr metody (**Parameter**)
  - **DisplayName** - zobrazovaný název
  - **DisplayOrder** - pořadí, ve kterém se parametr zobrazí
  - **DataType** - datový typ, který může ovlivnit zobrazování pole v UI
  - **Validators** - Validátory, které se na daný parametr uplatní. Pokud parametr není komplexní datový typ. V tomto případě se validátory nastavují pro příslušný datový kontrakt.
  - **DataContract/DataContractName** - název datového kontraktu, nebo datový kontrakt
- Pro každý datový kontrakt (**DataContractInfo**)
  - **Name** - jeho celý název
- Pro každé pole datového kontraktu (**Field**)
  - **Enabled** - jestli je povoleno a je tedy graficky reprezentováno ve výstupu
  - **DisplayName** - zobrazovaný název
  - **DisplayOrder** - pořadí, ve kterém se zobrazí
  - **DataType** - datový typ, který může ovlivnit zobrazování pole v UI
  - **Validators** - validátory, které se na dané pole uplatní

Zobrazovaným názvem je vždy myšlen řetězec, který se zobrazí v uživatelském rozhraní a může se lišit od názvu metody. Například chceme-li pro metodu „GetBooks“ zobrazit název „Seznam knih“, který bude na metodu odkazovat.

Pro vstupní parametr nebo návratový typ lze nastavit datový kontrakt, tedy pokud se jedná o komplexní datový typ. To lze dvěma způsoby. Buď datový kontrakt nastavíme v sekci DataContracts a nebo zvlášť pro každý vstupní parametr, nebo návratový typ. V prvním případě je nastavení společné pro všechny metody, které datový kontrakt používají, v druhém se nastavení aplikuje pouze na konkrétní položku.

Při čtení konfiguračního souboru pak má nastavení specifické pro konkrétní položku přednost před globálním nastavením.

Datový typ (DataType) slouží k bližší specifikaci toho, o jaká data se jedná. Hodnota pole je pouze předána vyšší vrstvě a knihovna AWS.Service.Proxy s ní nijak nepracuje.

Vyšší vrstva pak tyto informace může použít k úpravě generovaného grafického rozhraní. Atribut DataType může nabývat následujících hodnot:

- **Default** - výchozí nespecifikovaná hodnota
- **Custom** - vlastní datový typ
- **DateTime** - datum a čas
- **Date** - pouze datum
- **Time** - pouze čas
- **Duration** - časový úsek
- **PhoneNumber** - telefonní číslo
- **Currency** - peněžitá částka
- **Text** - obecný text
- **Html** - text, který má být interpretován jako HTML
- **MultilineText** - text na více řádků
- **EmailAddress** - emailová adresa
- **Password** - heslo
- **Url** - url adresa
- **ImageUrl** - url adresa obrázku

Důležitou vlastností konfigurace je, že nemusí přímo odpovídat definici služby. Dojde-li například k přidání nové metody do služby, není potřeba aktualizovat konfigurační soubor. Pokud totiž není daná metoda v souboru obsažena, program si na základě reflexe sám vytvoří danou část konfigurace. Protože je ale použita reflexe a k získání konfiguračních dat o metodě služby dochází velmi často, je tato část konfigurace vždy uložena do speciální Cache, aby nedošlo k zbytečnému zpomalování aplikace.

7.4 Validace

V kapitole o konfiguraci byly zmíněny validátory. Těch je k dispozici celkem 5 a jsou uplatnitelné na vstupní parametry metod. Validátor vždy funguje jen s jednoduchým datovým typem. Pokud je vstupní parametr komplexní třída, jsou validátory uplatňovány na veřejné vlastnosti této třídy a konfigurace validátoru probíhá v sekci datových kontraktů. Každý validátor má dva povinné parametry, které určují, jestli je validátor povolen a zprávu, která se zobrazí, jestliže validace neprojde. Další nastavení validátoru se pak již liší na základě jeho typu. Ten může být:

- **RequiredFieldValidator** – Kontroluje, zda vstupní řetězec není prázdný.
- **RangeValidator** – Kontroluje rozsah. Vstupní řetězec tedy musí být číselného typu a musí být ze zadaného rozsahu, který je možno upravit v konfiguračním souboru.
- **LengthValidator** – Kontroluje délku řetězce. Opět musí být ze zadaného rozsahu v konfiguračním souboru.
- **RegularExpressionValidator** – Validuje vstupní řetězec oproti regulérnímu výrazu, který je zadán v konfiguračním souboru
- **CompareValidator** – Validace projde úspěšně, pokud je předaná hodnota rovna buď hodnotě zadané v konfiguračním souboru, nebo hodnotě z jiného vyplněného pole v rámci vstupních parametrů nebo datového kontraktu. Typické použití je kontrola, zda uživatel zadal opakovaně stejné heslo při vytváření uživatelského účtu.

Tyto validátory pokrývají nejčastěji používané scénáře validace dat. Pokud je třeba složitější validace, která například komunikuje z databází, musí si ji vývojář naimplementovat ručně.

Jeden vstupní parametr může mít nakonfigurováno více validátorů. Protože ve většině případů ale chceme uživateli zobrazit ke každému vstupnímu poli vždy jen jednu chybovou hlášku, má každý validátor pevně přednastavenou prioritu vykonávání. Takto je zaručeno, že se vrátí vždy chybová zpráva prvního validátoru, který neprojde. Pořadí vykonávání validátorů je následující:

1. RequiredFieldValidator
2. LengthValidator
3. RangeValidator
4. RegularExpressionValidator
5. CompareValidator

8 Část MVC (AWS.Web.Core)

Knihovna AWS.Web.Core má za úkol zobrazovat grafické rozhraní. O samotné volání služby a získávání informací o jejích metodách se již nestará. Pouze udržuje spojení přes proxy třídu na službu, které pak předává nižší vrstvě. K zobrazování grafického rozhraní využívá ASP.NET MVC Framework.

8.1 Třída AWSController

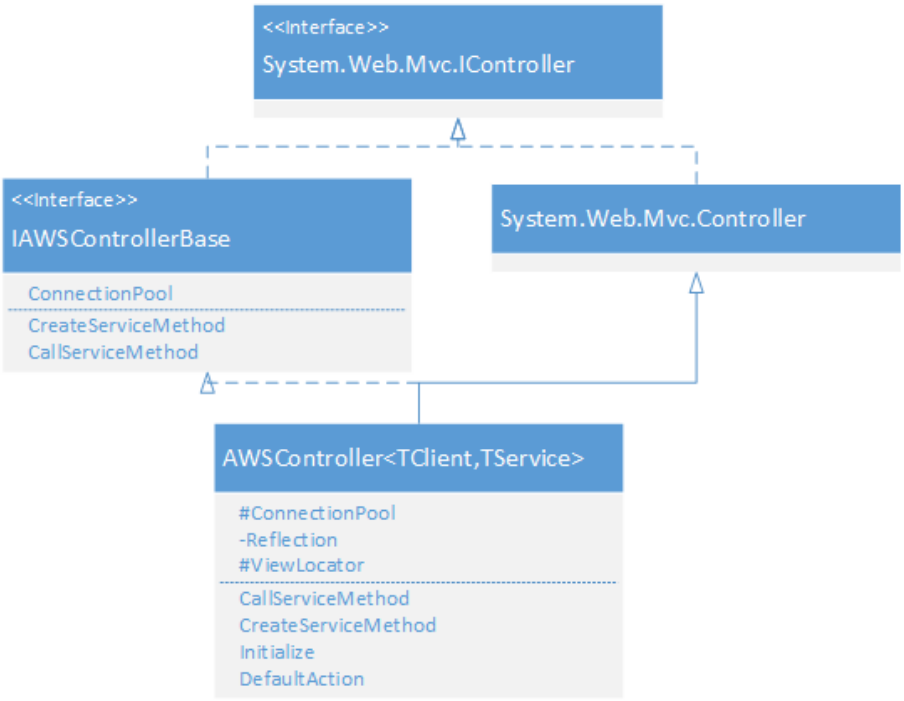
V kapitole 5.2 bylo řečeno, jak MVC Framework zpracovává požadavek. Prvním krokem, který musí vývojář udělat, je vytvořit třídu Controller a napsat akční metodu, která se vyvolá při požadavku na daný zdroj. Také bylo řečeno, že třída Controller musí dědit z System.Web.Mvc.Controller.

Podobně, pokud vývojář používá služby AWS Frameworku, musí jeho Controller dědit ze speciální třídy, která mu tyto služby poskytne. Tou třídou je AWS.Web.Core.AWSController. Jedná se o generickou třídu. A právě přes tyto generické typy je Frameworku předán typ služby, se kterou má Controller pracovat. Konkrétní Controller je tedy vždy omezen na využívání jedné služby. Různé AWSController třídy však mohou používat libovolné množství služeb.

Podíváme-li se na třídní diagram na obrázku 8, vidíme, že třída navíc implementuje rozhraní IAWSControllerBase. To umožňuje vývojáři napsat si celou vlastní implementaci této třídy bez porušení funkcionality zbytku Frameworku. Vidíme také, že třída má dvě chráněné vlastnosti (protected properties). Jsou jimi ConnectionPool, který má za úkol udržovat spojení na službu přes proxy třídu (detailně se mu věnuje kapitola 8.2) a ViewLocator. Ten je zodpovědný za výběr šablony pro zobrazení dat (viz. kapitola 8.6). Třída má dále dvě metody, jednu pro vytvoření reference na třídu služby a jednu pro samotné zavolání této metody.

Metoda pro volání metod služby (CallServiceMethod) přijímá název volané služby a kolekci NameValueCollection s dvojicemi název parametru a jeho hodnota. V kapitole 7.1 byl popsán způsob, jakým hodnoty parametrů přiřazují k volané instanci metody služby. V tomto procesu mohou nastat výjimky vzniklé při konverzi do konkrétního typu vstupního parametru. Ty jsou v těle metody zachyceny a předány do kolekci ModelState. Pokud tedy uživatel například do pole pro věk zadá textový řetězec, který neobsahuje pouze číslice, aplikace vyvolá výjimku. To je odchycena a projeví se ve View jako nevalidní pole.

Dále obsahuje inicializační metodu, ve které se nastavuje konkrétní implementace ViewLocator a také ActionInvoker. Inicializační metoda bude dále popsána v následujících kapitolách.



Obrázek 8: Třídní diagram AWSControlleru

8.1.1 Výchozí akční metody

AWSController obsahuje dvě výchozí akční metody, které jsou volány pokud není nalezena vlastní implementace akční metody.

Metoda pro obsloužení požadavku GET nejdříve zjistí informace o metodě služby, která se má zavolat z routovacích informací. Poté provede kontrolu, jestli je možné metodu vykonat. To lze v případě, pokud je volaná metoda služby bez argumentů, nebo má pouze jeden argument a je z routovacích informací dostupná hodnota "id". Pokud ano, dojde k volání metody. Nakonec je výsledek předán třídě ViewLocator, která se postará o vyhledání vhodného View.

Metoda pro obsloužení požadavku POST očekává, že příchozí požadavek obsahuje formulářová data. Ty jsou získána při bindování modelu. Modelem je pro nás instance třídy ServiceMethod s nastavenými hodnotami argumentů. Proto k bindování není možné použít výchozí ModelBinder. Akce tedy přijímá objekt typu AWSActionDescriptor, kterému přiřazuje speciální ModelBinder (AWSModelBinder). Ten získá informace o volané službě a na základě nich vytvoří instanci třídy ServiceMethod. Poté se pokusí přiřadit všechny hodnoty argumentů volané metody z ValueProvider (viz. kapitola 5.3.3). Pokud nastane výjimka při přetypování hodnot z formuláře do cílového typu argumentu služby, výjimku zachytí a přidá záznam do kolekce ModelState. Dále provede validaci vstupních dat vůči validátorům nastavených v konfiguračním souboru AWSFrameworku. Pokud validace selže opět vloží záznam do kolekce ModelState. Nakonec je vytvořena instance

třídy AWSActionDescriptor a je jí předán odkaz na metodu služby s nastavenými vstupními argumenty a kolekce validačních hlášení.

V těle akční metody dojde pouze k volání metody služby v případě že validace prošla a opět předání výsledku do ViewLocatoru pro výběr vhodného View.

8.2 Práce s připojením ke službě

Součástí AWSControlleru je ConnectionPool. Ten slouží k získávání instance proxy třídy pro volání metod webové služby. Tyto instance se ukládají ve statické struktuře a jsou tedy společné pro všechny požadavky na webový server. Pro každou definovanou službu si AWSFramework udržuje určitý počet otevřených spojení. Ten se dá nastavit při startu aplikace pro každou službu zvlášť. Defaultně je však nastaveno na 10. (viz. kapitola 8.3).

Vnitřně je pool implementován jako fronta. Pokud je vyžadováno spojení, je z fronty odebrán první prvek, jestliže je ale fronta prázdná vytvoří se nová instance proxy třídy. Po použití konkrétního spojení je proxy třída vrácena zpět do poolu. V případě, že fronta již obsahuje daný počet otevřených spojení, je uzavřeno a dojde k zahození instance proxy třídy. V opačném případě se pouze vrátí do fronty pro další použití.

Každá proxy třída služby dědí z abstraktní třídy System.ServiceModel.ClientBase<TChannel>, kde TChannel je třída pro spojení se službou (této problematice se věnovala kapitola 3.1). Tato třída má několik konstruktorů. Jeden je vždy bez parametrů a dalšími lze ovlivnit chování a konfiguraci proxy třídy a to:

- nastavení koncového bodu služby
- nastavení adresy koncového bodu služby
- název konfigurace služby v konfiguračním souboru aplikace, která se použije
- vazba služby (Binding)
- nastavení instance zpětného volání (InstanceContext).

Vývojářům, kteří AWS Framework používají, musí být umožněno vytvářet instance proxy třídy služby jakýmkoliv způsobem. Díky použití ConnectionPoolu se ale instance vytváří uvnitř frameworku, až když je potřeba. Aby byla dosažena možnost vytvářet instance libovolným způsobem, byl použit návrhový vzor Inversion of Control.

Návrhový vzor Inversion of Control (dále jen IoC) umožňuje uvolnit vztahy mezi jinak těsně svázanými komponentami. V klasickém modelu programování si třída sama vytváří instance jiných tříd, které používá. S použitím IoC naopak třídě předáváme tuto instanci například s použitím rozhraní. Zbavíme se tak závislosti na konkrétní implementaci předávané třídy.

Příkladem v .NET Frameworku může být třída System.IO.BufferedStream. Ta si v interním bufferu uchovává data a zapíše je do streamu až, když se tento buffer naplní. Jaký stream se použije, je předáno v konstruktoru této třídy. Můžeme jí předat například FileStream nebo MemoryStream. Třída tedy není závislá na konkrétním streamu a používá jej jen přes abstraktní rozhraní.

Nastavení pravidel vytváření instance proxy třídy se děje při startu aplikace, kterému je věnována následující kapitola.

8.3 Nastavení AWS Frameworku při startu aplikace

Vývojář má možnost při startu aplikace definovat, které služby využívá. Není to podmínkou pro správnou funkčnost frameworku, ale přináší to určité výhody:

- Registrované služby se automaticky přidají do konfiguračního souboru (aws.config). Pokud soubor neexistuje, pak je vytvořen s defaultními hodnotami.
- Je možné ovlivnit chování poolu spojení na službu a to určením maximálního počtu udržovaných aktivních připojení pro jednotlivé služby.
- Lze určit funkci, která se použije při vytváření nové instance proxy třídy.

Registrace služby probíhá v souboru Global.asax. To je soubor, který může obsahovat kterákoliv ASP.NET webová aplikace. Musí být v kořenovém adresáři. Interně se jedná o třídu, která dědí přímo z System.Web.HttpApplication a umožňuje navázat kód na různé události, které v průběhu životního cyklu aplikace nastávají. Mezi nejdůležitější v těchto událostech patří:

- **Application_Init:** Je vyvolána při inicializaci aplikace.
- **Application_Start:** Je vyvolána při startu aplikace.
- **Session_Start:** Je vyvolána, když je vytvořena nová session mezi webovým serverem a klientem.
- **Application_BeginRequest:** Je vyvolána pokaždé, když přijde nový požadavek.
- **Application_EndRequest:** Je vyvolána po zpracování požadavku.
- **Application_AuthenticateRequest:** Nastane, pokud je vyžadována autentizace.
- **Application_Error:** Nastane, pokud je v aplikaci vyvolána neočekávaná výjimka.
- **Session_End:** Nastane po ukončení session mezi serverem a klientem.
- **Application_End:** Je vyvolána před ukončením aplikace.

Pro nás je důležitá událost Application_Start. Tu využívá i samotný MVC Framework například pro registraci routování nebo pro nastavení css stylů a souborů s javascriptem, které budou předávány v každém požadavku.

Bývá zvykem umísťovat třídy, jejichž metody se mají spustit při startu aplikace do systémové složky App_Start. Podobně jako například složka App_Data, nebo App_Code není přístupná přes HTTP požadavek z webu. Metody těchto tříd bývají statické a volají se přímo z Application_Start v souboru Global.asax.

Registraci služby pak lze provést voláním statické metody RegisterService třídy AWS.Web.Core.AWSWebCore. Příklad je ve výpisu kódu 6.

```
public class AWSWebCoreConfig
{
    public static void RegisterServices()
    {
      AWSWebCore.RegisterService<BookStoreServiceClient, BookStoreService>(10);
    }
}
```

Výpis 6: Přidání odkazu na službu při startu aplikace

Vidíme, že metoda je generická a jako datové typy ji předáváme proxy třídu a rozhraní Channel (viz. kapitola 3.1). Tím zajistíme, že Framework je schopen vytvořit novou instanci proxy třídy, když ji potřebuje a to zavoláním jejího defaultního konstruktoru.

Pokud je ale potřeba instanci proxy třídy vytvořit jinak, musíme postupovat obdobně, jako v kódu 7. Stačí implementovat rozhraní IServiceClientFactory, které předepisuje pouze jednu metodu Create. Ta je bez parametrů a musí vracet instanci proxy třídy. Pokud takovouto třídu máme, lze její typ předat jako třetí generický parametr metodě RegisterService.

Obě zmíněné metody RegisterService jsou přetíženy a jako svůj parametr mohou přijímat maximální počet aktivních spojení na službu.

```
public class AWSWebCoreConfig
  {
    public static void RegisterServices()
    {
      AWSWebCore.RegisterService<BookStoreServiceClient, BookStoreService,
          MyServiceClientFactory>(10);
    }
  }

  class MyServiceClientFactory : AWS.Web.Core.Connectivity.IServiceClientFactory<
      BookStoreServiceClient, BookStoreService>
  {

    public BookStoreServiceClient Create()
    {
      return new BookStoreServiceClient("MyEndpointConfigurationName");
    }
  }
```

Výpis 7: Přidání odkazu na službu s vlastním kódem pro vytvoření instance proxy třídy při startu aplikace

8.4 Routování v AWS frameworku

AWS Framework nijak nezasahuje do routování požadavků tak, jak bylo popsáno v kapitole 5.3.1 v MVC Frameworku. Pouze umožňuje zavolat defaultní akční metodu, pokud není žádná nalezena ve třídě Controller. Podmínkou pro toto chování je, aby daný Controller dědil z třídy AWSController. Poté v případě nenalezení vhodné akční metody je

zavolána metoda DefaultAction bázové třídy. Toho je docíleno použitím vlastní implementace třídy ControllerActionInvoker, která je nastavena v inicializační metodě třídy AWSController. Dále si v routovacích datech uchovává hodnotu "methodName", která obsahuje jméno aktuálně volané metody služby. Ta je nastavena při vykonávání metody FindAction třídy AWSControllerActionInvoker.

8.5 Šablony a Precompiled Views

Podíváme-li se na diagram na obrázku 6, vidíme že AWS Framework umožňuje volat již předpřipravené View neboli šablony a to v případě, že vývojář nedefinoval své vlastní View. Tyto šablony jsou obsaženy přímo v knihovně AWS.Web.Core.dll a jsou čtyř typů:

- **InputForm.cshtml** - formulář pro vstup dat, který očekává od uživatele nějaké hodnoty
- **OutputForm.cshtml** - formulář, který je pouze pro čtení
- **OutputGrid.cshtml** - tabulka pouze pro čtení
- **Menu.cshtml** - základní menu

Na každé View se dá v ASP.NET MVC odkazovat pomocí takzvané virtuální cesty (PageVirtualPath). Ta je relativní vzhledem ke kořenové složce aplikace a má tvar například `~/Views/MyView.cshtml`. Cesta ke všem defautním šablonám začíná `~/Views/Shared/AWSTemplates/Default`.

Soubory cshtml, ale nejsou třídy a podobně jak tomu je s klasickými ASP.NET stránkami (aspx), jsou převedeny do třídy a poté zkompilovány až za běhu programu. Konkrétně při prvním požadavku na webový server. Šablony, které jsou obsaženy v AWSFrameworku museli být ale zkompilovány již předem. K tomu byl použit nástroj RazorGenerator ([6]). Ten právě ze souboru cshtml vygeneruje třídu. Kód 8 reprezentuje stejné View, jaké bylo použito pro ukázku v kódu 2. Virtuální cesta je definována v atributu PageVirtualPathAttribute, kterým je třída dekorována.

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("RazorGenerator", "1.5.4.0")]
  [System.Web.WebPages.PageVirtualPathAttribute("~/Views/Shared/AWSTemplates/Default/
      MyView.cshtml")]
  public partial class MyView : AWS.Web.Core.AWSWebViewPage<dynamic>
  {
    public MyView()
    {
    }
    public override void Execute()
    {
      WriteLiteral ("<h2>Index</h2>\r\n");
      if (Model != null)
      {
        WriteLiteral ("<b>Hello");
        Write(Model);
        WriteLiteral ("!</b>\r\n");
```

```
        }
    }
}
```

Výpis 8: Příklad předkompilovaného View

Aby vše korektně fungovalo je nutné předkompilované View registrovat ve webové aplikaci. To se děje automaticky v assembly AWS.Web.Core při jejím startu tím, že se do kolekce ViewEngines (System.Web.Mvc.ViewEngines) přidá další ViewEngine typu PrecompiledMvcEngine. Ten je zaregistrovaný až za defaultním, takže je možné ve webové aplikaci předkompilované View přepisovat vlastními, tak že je vložíme do stejné složky.

8.6 Algoritmus výběru šablony a jak jej ovlivnit

Šablony byly diskutovány v předešlé kapitole. Jejich nejčastější použití je, když se zavolá defaultní akční metoda v Controlleru. Vývojář je však může používat i ve vlastních View. Jak ale AWS Framework pozná, kterou šablonu má použít?

K tomuto účelu existuje třída AWSViewLocator. Její instance je vytvořena vždy při inicializaci třídy AWSController (metoda Initialize) a je přiřazena do vlastnosti ViewLocator. Nejdůležitější metodou ve FindViewPath, které jsou předány veškeré informace o volané akční metodě a jejím úkolem je vybrat vhodnou šablonu. Tato metoda je deklarována jako virtuální a může být tedy přepsána ve třídě, která z AWSViewLocator dědí. Takto je vývojářům umožněno ovlivnit výběr šablony podle sebe. Mohou si také napsat své vlastní šablony a logiku, která je za určitých okolností spustí.

8.7 Speciální Helpery AWS Frameworku

Pro generování grafického rozhraní je použit koncept pomocných metod (Helper Methods). Ten byl představen v kapitole 5.5.

Každé View musí dědit z třídy WebViewPage nebo z třídy, která je z ní odvozená. WebViewPage má vlastnost Html, která odkazuje na instanci HtmlHelperu. To je také základní a nejčastěji používaný helper. AWSHelper tuto třídu rozšiřuje tak, že přidává metodu Aws(), která vrátí instanci AWSHelperu, přes kterou je možné generovat uživatelské rozhraní. Na obrázku 9 je zobrazena struktura závislostí v této třídě.

Z diagramu jde vidět, že se třída skládá z množství přetížených metod, které slouží ke generování HTML kódu a několika pomocných metod a vlastností. Předpokládá se, že pokud bude vývojář psát své vlastní View bude třídu AWSHelper využívat ve svém kódu a dále ji rozšiřovat pomocí extension methods. Proto si ji podrobněji popíšeme.

V kapitole 7.1 bylo řečeno že se vstupní a výstupní parametry metod služby dělí na jednoduché datové typy a datové kontrakty. Takto s nimi pracuje i AWSHelper. Jednoduché datové typy si ale ještě rozděluje na podskupiny podle toho, jak budou zobrazeny uživateli. A to následovně:

- Řetězcové typy - String
- Boolean - bool

- Číselné datové typy - byte, sbyte, decimal, double, Int16,Iint32, Int64, UInt16, UInt32,UInt64
- Datum - Datetime
- Enumerátor - Enum

Generování HTML kódu je v helperech dále ovlivněno nastaveným polem DataType v konfiguračním souboru (viz. kapitola 7.3), který je metodám předán.

8.7.1 Prvky InputField a InputForm

Tyto dvě metody slouží výhradně k zadání vstupu od uživatele. Vstupní pole (InputField) je jedno pole, které reprezentuje jednoduchý datový typ. Na základě toho, o jaký typ se jedná je vygenerován příslušný ovládací prvek.

- **Řetězcové typy** - Vstupní pole (TextBox), pokud je v konfiguraci nastaven atribut DataType na Password jsou hodnoty textového pole skryty.
- **Boolean** - Zaškrkávací políčko (CheckBox).
- **Číselné datové typy** - Vstupní pole (TextBox).
- **Datum** - Datetime - Je-li na stránce k dispozici knihovna JQueryUI, tak se vygeneruje jako speciální datumové pole (DateTimePicker), pokud k dispozici není, tak jako obyčejné vstupní pole (TextBox).
- **Enumerátor** - Enum - Rolovací nabídka se všemi hodnotami, které enumerátor nabízí (DropDownList).

Metoda InputField je přetížena, aby umožnila vývojářům snadnější použití. V nejobecnější implementaci obsahuje tyto parametry:

- **fieldName (string)** - název vstupní hodnoty
- **fieldId (string)** - id vstupní hodnoty, které pole jednoznačně identifikuje v rámci formuláře
- **fieldValue (string)** - výchozí hodnota
- **fieldType (Type)** - typ vstupního pole
- **dataType (AWSDataType)** - datový typ vstupního pole (slouží k ovlivnění HTML výstupu)
- **enableValidation (AWSDataType)** - indikuje, zda generovat validační zprávy
- **fieldNameHtmlAttributes (IDictionary<string, object>)** - kolekce HTML atributů, které se přidají k názvu pole

- **fieldValueHtmlAtributes (IDictionary<string, object>)** - kolekce HTML atributů, které se přidají k hodnotě pole

HTML výstup jednoho vstupního pole pak může vypadat takto:

```
<label>Nazev knihy:</label>
<input id="AWS_Demo_BookStoreServiceReference_Book_Title" type="text" value="" name="
    AWS_Demo_BookStoreServiceReference_Book_Title">
<span class="field-validation-valid" data-valmsg-replace="true" data-valmsg-for="Nazev␣
    knihy"></span>
```

Výpis 9: Příklad výstupu AWSHelper s validací

Metody, které generují vstupní pole nejsou, závislé na AWS Frameworku a mohou být použity i samostatně. Naproti tomu vstupní formulář (InputForm) již přímo pracuje s metodami služby. A je mu pouze předán název metody, pro kterou vygeneruje formulář pro vstupní parametry této metody k čemuž právě InputField použije.

Poté, co klient vyplní formulář, je zpět odeslán na server. Data z tohoto formuláře jsou získána díky jedinečnému identifikátoru, který je přiřazen každému vstupnímu poli.

8.7.2 Prvky OutputField a OutputForm

Obdobně jako pro vstupní formulář a pole existují helper metody i pro výstupní. Tím je typicky návratová hodnota volání metody služby. Výstupní pole se zobrazují pouze pro čtení a neumožňují editaci záznamů. Podoba vygenerovaného prvku závisí na nastavení atributu DataType v konfiguraci.

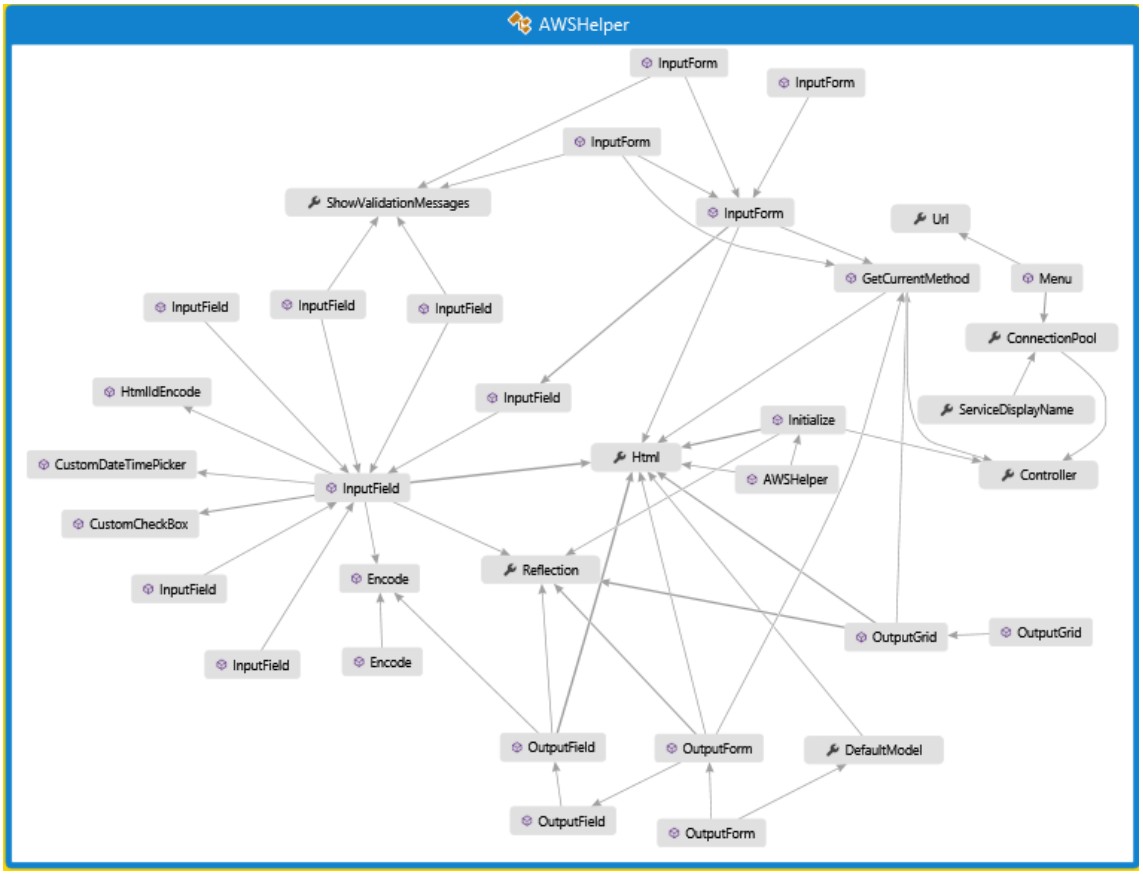
- **DateTime** - datum a čas ve formátu d.M.yyyy HH:mm:ss
- **Date** - pouze datum ve formátu d.M.yyyy
- **Time** - pouze čas ve formátu HH:mm:ss
- **Html** - HTML text
- **EmailAddress** - odkaz na emailovou adresu
- **Url** - odkaz na zadanou URL adresu
- **ImageUrl** - obrázek na zadané url

V ostatních případech se vypíše prostý text. Pouze pokud je datovým typem Boolean vygeneruje se check box. Opět platí že OutputField je obecný bez návaznosti na AWSFramework a OutputForm se vždy vytváří v kontextu nějaké metody služby.

8.7.3 Prvek Menu

Posledním prvkem, který je v AWSHelperu k dispozici, je Menu. To vygeneruje na základě služby, kterou třída Controller používá, seznam všech dostupných metod (Ten lze

upravovat v konfiguračním souboru). Každá položka menu pak odkazuje na stránku s konkrétní funkcionalitou na základě definice této metody.



Obrázek 9: Diagram závislostí ve třídě AWSHelper

9 Vzorová aplikace

Součástí řešení je ukázková aplikace, která demonstruje funkcionalitu a možnosti nastavení AWS Frameworku. Je složena ze služby a webového rozhraní, které reprezentuje metody této služby. Ukázka je koncipována do jednoduchého objednávkového systému knih s možností registrace zákazníků. Cílem nebylo vytvořit aplikaci pro produkční prostředí, pouze demonstrovat vyvíjený framework. Příloha B obsahuje snímky obrazovky z ukázkové aplikace.

Projekt se službou (AWS.Demo.Service) běží na výchozím portu 19654. Adresa služby je `~/BookStoreService.svc`. Data nejsou ukládána do žádného persistentního úložiště, pouze do cache paměti serveru. Proto jsou nové záznamy po čase vymazány.

Služba obsahuje metody pro:

- Přihlášení a odhlášení uživatele.
- Získání seznamu všech knih, nebo detailu konkrétní knihy.
- Získání seznamu registrovaných zákazníku a objednávek přihlášeného zákazníka.
- Vložení nové knihy a registraci nového zákazníka.
- Vložení knihy do košíku.
- Dokončení objednávky, kdy se vyprázdní košík a je vytvořena nová objednávka pro přihlášeného zákazníka.

Samotné webové rozhraní (projekt AWS.Demo) nabízí dvě ukázky:

- **Základní nastavení** - Účelem této ukázky je demonstrovat aplikaci, která vznikla pouze použitím konfiguračního souboru. Controller pro tuto ukázku se jmenuje BasicController a v jeho těle je pouze odkaz na službu, která se má využívat. Veškeré další nastavení je pak pouze v souboru aws.config. Ten mění názvy metod a jejich parametrů do češtiny, dále přidává validaci na tyto parametry a upravuje zobrazovaná pole datových kontraktů.
- **Pokročilé nastavení** - Úvodní stránka demonstruje kompletně vlastní implementaci. Pro horní menu je vytvořen statický partial view (`~/Views/Shared/AWSTemplates/_Menu.cshtml`). Akční metoda Index volá metodu služby pro získání seznamu knih. Její výsledek pak předává View, které je také implementováno ručně. View využívá metod AWSHelperu, zobrazuje menu a tabulku se seznamem knih do které přidává odkaz na detail knihy. Dole pak je tlačítko pro dokončení objednávky.
  Detail knihy ukazuje situaci, kdy je celý proces až po zobrazení výsledku nechán na frameworku. Je tedy zavolána defaultní akční metoda, která získá detail knihy podle id v URL. ViewLocator je upraven tak, aby pro tuto akční metodu vybral View `~/Views/Shared/AWSTemplates/BookDetail.cshtml`. Protože je tato šablona použita jen pro konkrétní jednu metodu služby, může být silně typována.

Víme totiž, že modelem pro ni bude instance třídy Book. Ve View pak je možné vygenerovat jakékoliv grafické rozhraní.
Metoda pro objednávku knihy přijímá dva parametry. Id objednávané knihy a její množství. V Controlleru je akční metoda pro objednání knihy přepsána tak, že id knihy převezme z URL a počet nastaví na jedna. Poté knihu objedná zavoláním metody "OrderBook". V poslední řadě získá seznam všech knih a zobrazí domovskou stránku (view Index).
Nakonec je přepsána defaultní šablona pro výstupní formulář. Jednoduše je ve složce `~/Views/Shared/AWSTemplates/Default` nahrazena vlastní implementací, která vždy na konci formuláře zobrazí odkaz zpět na hlavní stránku.

10 Závěr

Cílem této diplomové práce bylo prozkoumat možnosti ASP.NET MVC Frameworku a navrhnout vlastní aplikační rámec, který by zjednodušoval vývoj webových aplikací založených na službách.

Navržený rámec (AWS Framework) funguje nad jakoukoliv definicí SOAP webové služby a umožňuje vývojářům snadné použití a vygenerování grafického rozhraní. Byl kladen důraz na to, aby všechna rozšíření dodržovala konvence zavedené v MVC Frameworku a zjednodušil se pak případný další rozvoj projektu.

Funkcionalita frameworku je demonstrována ve vzorové aplikaci, která je k dispozici na přiloženém CD.

11 Reference

[1] *Securely Implement Request Processing, Filtering, and Content Redirection with HTTP Pipelines in ASP.NET*[Online] Dostupné z:
http://msdn.microsoft.com/en-us/magazine/cc188942.aspx

[2] *An Introduction to ASP.NET MVC Extensibility*[Online] Dostupné z:
https://www.simple-talk.com/content/article.aspx?article=1358

[3] Matthew MacDonald, Mario Szpuszta
ASP.NET 3.5 a C# 2008 tvorba dynamických stránek profesionálně, Zoner Press, 2008

[4] *ASP.NET MVC 3 RTM Source code* [Online] Dostupné z:
http://aspnet.codeplex.com/releases/view/58781

[5] *ServiceModel Metadata Utility Tool (Svcutil.exe)*[Online] Dostupné z:
http://msdn.microsoft.com/en-us/library/aa347733.aspx

[6] *Razor Generator*[Online] Dostupné z:
http://razorgenerator.codeplex.com/

[7] *ASP.NET MVC Pipeline*[Online] Dostupné z:
http://waldyrfelix.net/download/aspnet-mvc-pipeline.pdf

[8] *Scott Guthrie Blog - Introducing "Razor"*[Online] Dostupné z:
http://weblogs.asp.net/scottgu/archive/2010/07/02/introducing-razor.aspx

A XSD Schéma konfiguračního souboru AWS Frameworku

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="Configuration"
    targetNamespace="http://aws.dynns.com/Configuration.xsd"
    elementFormDefault="qualified"
    xmlns="http://aws.dynns.com/Configuration.xsd"
    xmlns:mstns="http://aws.dynns.com/Configuration.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
  <xs:element name="Configuration" type="Configuration">
  </xs:element>

  <xs:complexType name="Configuration">
    <xs:sequence>
      <xs:element name="Service" type="Service" minOccurs="0" maxOccurs="unbounded"></
          xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Service">
    <xs:sequence>
      <xs:element name="EndPoint" minOccurs="1" maxOccurs="1">
        <xs:complexType>
          <xs:attribute  name="Address" type="xs:string"></xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:element name="Methods" minOccurs="1" maxOccurs="1" type="ServiceMethods"></
          xs:element>
      <xs:element name="DataContracts" minOccurs="1" maxOccurs="1" type="DataContracts"><
          /xs:element>
    </xs:sequence>
    <xs:attribute  name="AssamblyName" type="xs:string"></xs:attribute>
    <xs:attribute  name="Name" type="xs:string"></xs:attribute>
    <xs:attribute  name="DisplayName" type="xs:string"></xs:attribute>
  </xs:complexType>

  <xs:complexType name="ServiceMethods">
    <xs:sequence>
      <xs:element name="MethodInfo" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="InputParameters" minOccurs="1" maxOccurs="1" nillable="false">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Parameter" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element name="Validators" minOccurs="0" maxOccurs="1" type="
                            Validators" nillable="true"></xs:element>
                        <xs:element name="DataContract" minOccurs="0" maxOccurs="1" type="
                            DataContractInfo"></xs:element>
                      </xs:sequence>
```

```
            <xs:attribute  name="Name" type="xs:string"></xs:attribute>
            <xs:attribute  name="DisplayName" type="xs:string" default=""></
                xs:attribute>
            <xs:attribute  name="DisplayOrder" type="xs:int" default="0"></xs:attribute
                >
            <xs:attribute  name="Type" type="xs:string"></xs:attribute>
            <xs:attribute  name="DataContractName" type="xs:string"></xs:attribute>
            <xs:attribute  name="DataType" default="Default">
              <xs:simpleType>
                < xs:restriction  base="xs:string">
                  <xs:enumeration value="Default"></xs:enumeration>
                  <xs:enumeration value="Custom"></xs:enumeration>
                  <xs:enumeration value="DateTime"></xs:enumeration>
                  <xs:enumeration value="Date"></xs:enumeration>
                  <xs:enumeration value="Time"></xs:enumeration>
                  <xs:enumeration value="Duration"></xs:enumeration>
                  <xs:enumeration value="PhoneNumber"></xs:enumeration>
                  <xs:enumeration value="Currency"></xs:enumeration>
                  <xs:enumeration value="Text"></xs:enumeration>
                  <xs:enumeration value="Html"></xs:enumeration>
                  <xs:enumeration value="MultilineText"></xs:enumeration>
                  <xs:enumeration value="EmailAddress"></xs:enumeration>
                  <xs:enumeration value="Password"></xs:enumeration>
                  <xs:enumeration value="Url"></xs:enumeration>
                  <xs:enumeration value="ImageUrl"></xs:enumeration>
                </ xs:restriction >
              </xs:simpleType>
            </ xs:attribute >
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="ReturnType" minOccurs="0" maxOccurs="1">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="DataContract" minOccurs="0" maxOccurs="1" type="
            DataContractInfo"></xs:element>
      </xs:sequence>
      <xs:attribute  name="IsArray" type="xs:boolean" default="false">
      </ xs:attribute >
      <xs:attribute  name="Type" type="xs:string"></xs:attribute>
      <xs:attribute  name="DataType" default="Default">
        <xs:simpleType>
          < xs:restriction  base="xs:string">
            <xs:enumeration value="Default"></xs:enumeration>
            <xs:enumeration value="Custom"></xs:enumeration>
            <xs:enumeration value="DateTime"></xs:enumeration>
            <xs:enumeration value="Date"></xs:enumeration>
            <xs:enumeration value="Time"></xs:enumeration>
            <xs:enumeration value="Duration"></xs:enumeration>
            <xs:enumeration value="PhoneNumber"></xs:enumeration>
            <xs:enumeration value="Currency"></xs:enumeration>
            <xs:enumeration value="Text"></xs:enumeration>
```

```
                    <xs:enumeration value="Html"></xs:enumeration>
                    <xs:enumeration value="MultilineText"></xs:enumeration>
                    <xs:enumeration value="EmailAddress"></xs:enumeration>
                    <xs:enumeration value="Password"></xs:enumeration>
                    <xs:enumeration value="Url"></xs:enumeration>
                    <xs:enumeration value="ImageUrl"></xs:enumeration>
                  </ xs:restriction >
                </xs:simpleType>
              </ xs:attribute >
              <xs:attribute  name="DataContractName" type="xs:string"></xs:attribute>
              <xs:attribute  name="RedirectAction" type="xs:string" default=""></xs:attribute>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute  name="Name" type="xs:string"></xs:attribute>
        <xs:attribute  name="DisplayName" type="xs:string" default=""></xs:attribute>
        <xs:attribute  name="Enabled" type="xs:boolean" default="true"></xs:attribute>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="DataContracts">
  <xs:sequence>
    <xs:element name="DataContractInfo" type="DataContractInfo" minOccurs="0" maxOccurs="
        unbounded"></xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="DataContractInfo">
  <xs:sequence>
    <xs:element name="Field" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Validators" minOccurs="0" maxOccurs="1" type="Validators"
              nillable="true"></xs:element>
        </xs:sequence>
        <xs:attribute  name="Name" type="xs:string"></xs:attribute>
        <xs:attribute  name="DisplayName" type="xs:string" default=""></xs:attribute>
        <xs:attribute  name="DisplayOrder" type="xs:int" default="0"></xs:attribute>
        <xs:attribute  name="Enabled" type="xs:boolean" default="true"></xs:attribute>
        <xs:attribute  name="Type" type="xs:string"></xs:attribute>
        <xs:attribute  name="Nullable" type="xs:boolean" default="false"></xs:attribute>
        <xs:attribute  name="DataType" default="Default">
          <xs:simpleType>
            < xs:restriction  base="xs:string">
              <xs:enumeration value="Default"></xs:enumeration>
              <xs:enumeration value="Custom"></xs:enumeration>
              <xs:enumeration value="DateTime"></xs:enumeration>
              <xs:enumeration value="Date"></xs:enumeration>
              <xs:enumeration value="Time"></xs:enumeration>
              <xs:enumeration value="Duration"></xs:enumeration>
              <xs:enumeration value="PhoneNumber"></xs:enumeration>
              <xs:enumeration value="Currency"></xs:enumeration>
```

```
                <xs:enumeration value="Text"></xs:enumeration>
                <xs:enumeration value="Html"></xs:enumeration>
                <xs:enumeration value="MultilineText"></xs:enumeration>
                <xs:enumeration value="EmailAddress"></xs:enumeration>
                <xs:enumeration value="Password"></xs:enumeration>
                <xs:enumeration value="Url"></xs:enumeration>
                <xs:enumeration value="ImageUrl"></xs:enumeration>
              </ xs:restriction >
            </xs:simpleType>
          </ xs:attribute >
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute  name="Name" type="xs:string"></xs:attribute>
</xs:complexType>

<xs:complexType name="Validators">
  <xs:sequence>
    <xs:element name="RequiredFieldValidator" minOccurs="0" maxOccurs="1" nillable="true">
      <xs:complexType>
        <xs:attribute  name="Enabled" type="xs:boolean" use="required" ></xs:attribute>
        <xs:attribute  name="Message" type="xs:string" use="optional" default="
            RequiredFieldValidator" ></xs:attribute>
      </xs:complexType>
    </xs:element>

    <xs:element name="RangeValidator" minOccurs="0" maxOccurs="1" nillable="true">
      <xs:complexType>
        <xs:attribute  name="Enabled" type="xs:boolean" use="required" ></xs:attribute>
        <xs:attribute  name="Message" type="xs:string" default="RangeValidator" ></xs:attribute
            >
        <xs:attribute  name="MinValue" type="xs:int" use="required"></xs:attribute>
        <xs:attribute  name="MaxValue" type="xs:int" use="required"></xs:attribute>
      </xs:complexType>
    </xs:element>

    <xs:element name="LengthValidator" minOccurs="0" maxOccurs="1" nillable="true">
      <xs:complexType>
        <xs:attribute  name="Enabled" type="xs:boolean" use="required"></xs:attribute>
        <xs:attribute  name="Message" type="xs:string" default="LengthValidator" ></
            xs:attribute>
        <xs:attribute  name="MinLength" type="xs:int" default="0"></xs:attribute>
        <xs:attribute  name="MaxLength" type="xs:int" default="0"></xs:attribute>
      </xs:complexType>
    </xs:element>

    <xs:element name="RegularExpressionValidator" minOccurs="0" maxOccurs="1" nillable="
        true">
      <xs:complexType>
        <xs:attribute  name="Enabled" type="xs:boolean" use="required" ></xs:attribute>
        <xs:attribute  name="Message" type="xs:string" default="RegularExpressionValidator" >
            </xs:attribute>
        <xs:attribute  name="Expression" type="xs:string" default=""></xs:attribute>
      </xs:complexType>
```

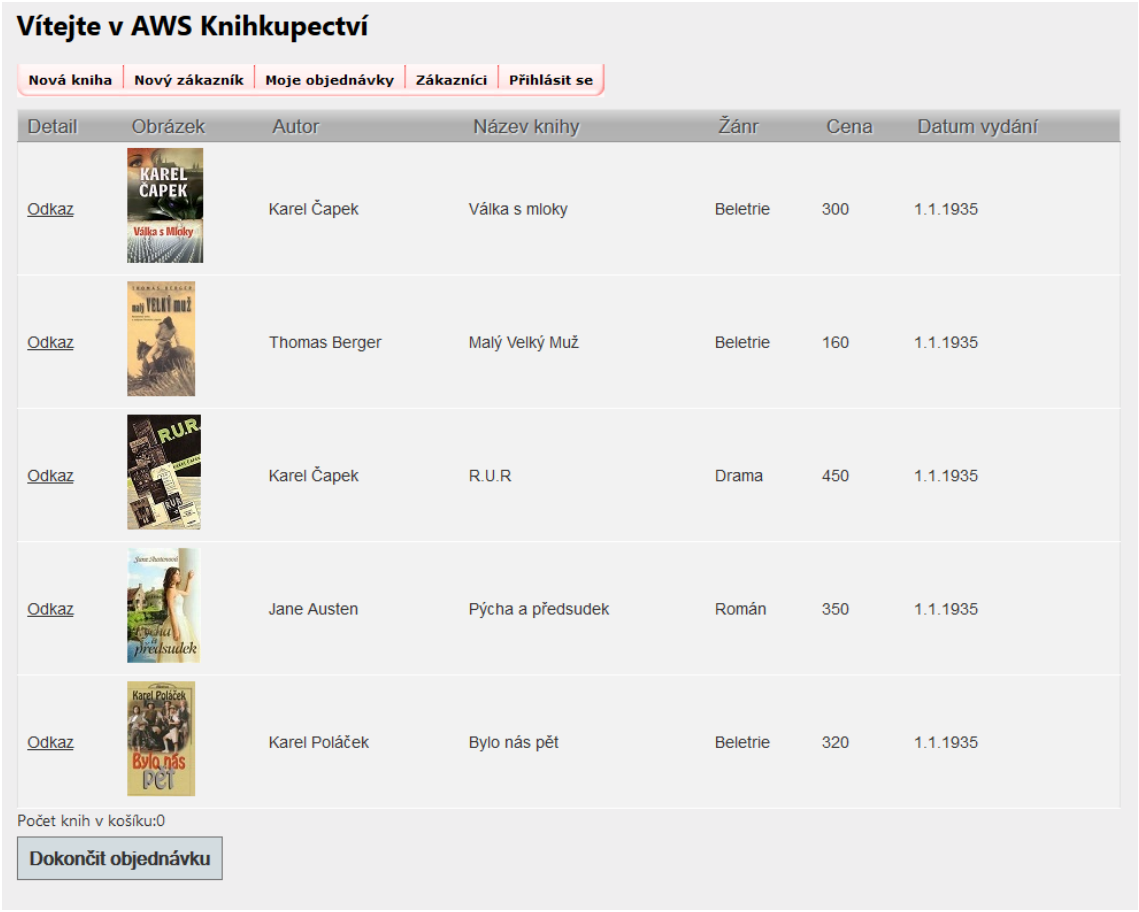
```
        </xs:element>

        <xs:element name="CompareValidator" minOccurs="0" maxOccurs="1" nillable="true" >
          <xs:complexType>
            <xs:attribute  name="Enabled" type="xs:boolean" use="required" ></xs:attribute>
            <xs:attribute  name="Message" type="xs:string" default="CompareValidotor" ></
                xs:attribute>
            <xs:attribute  name="CompareToFieldName" use="optional" default="" type="xs:string">
                </xs:attribute>
            <xs:attribute  name="CompereToValue" use="optional" default="" type="xs:string"></
                xs:attribute>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
```

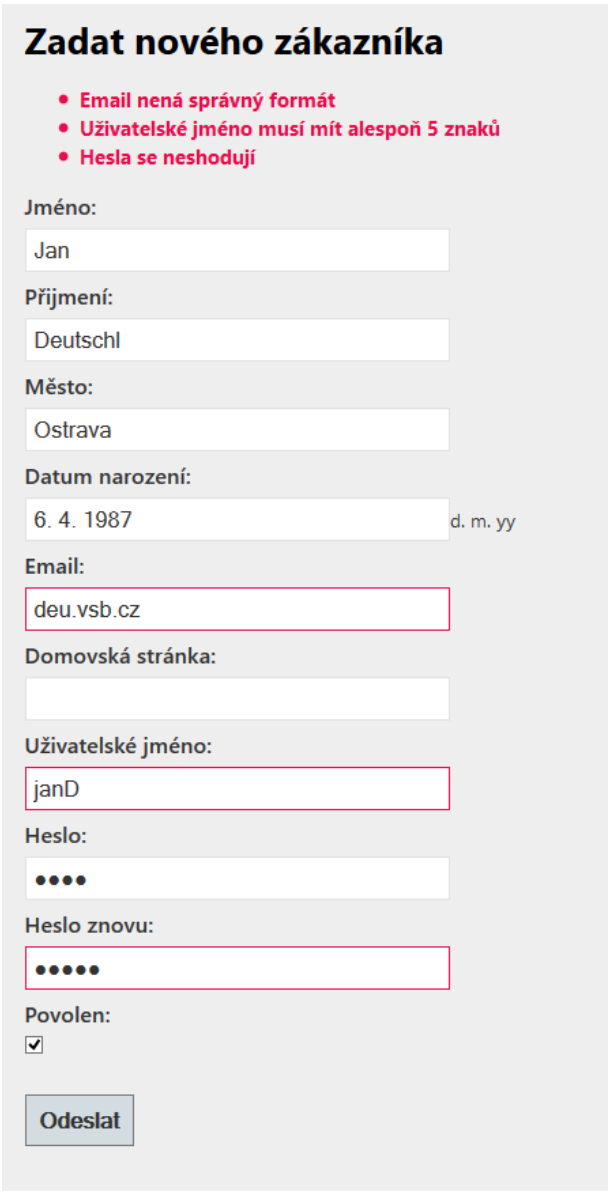
Výpis 10: XSD schéma konfiguračního souboru

B Ukázka vzorové aplikace

Obrázek 10: Detail knihy s možností objednávky



Obrázek 11: Úvodní stránka pokročilé ukázky



Obrázek 12: Formulář pro zadání nového zákazníka s validací